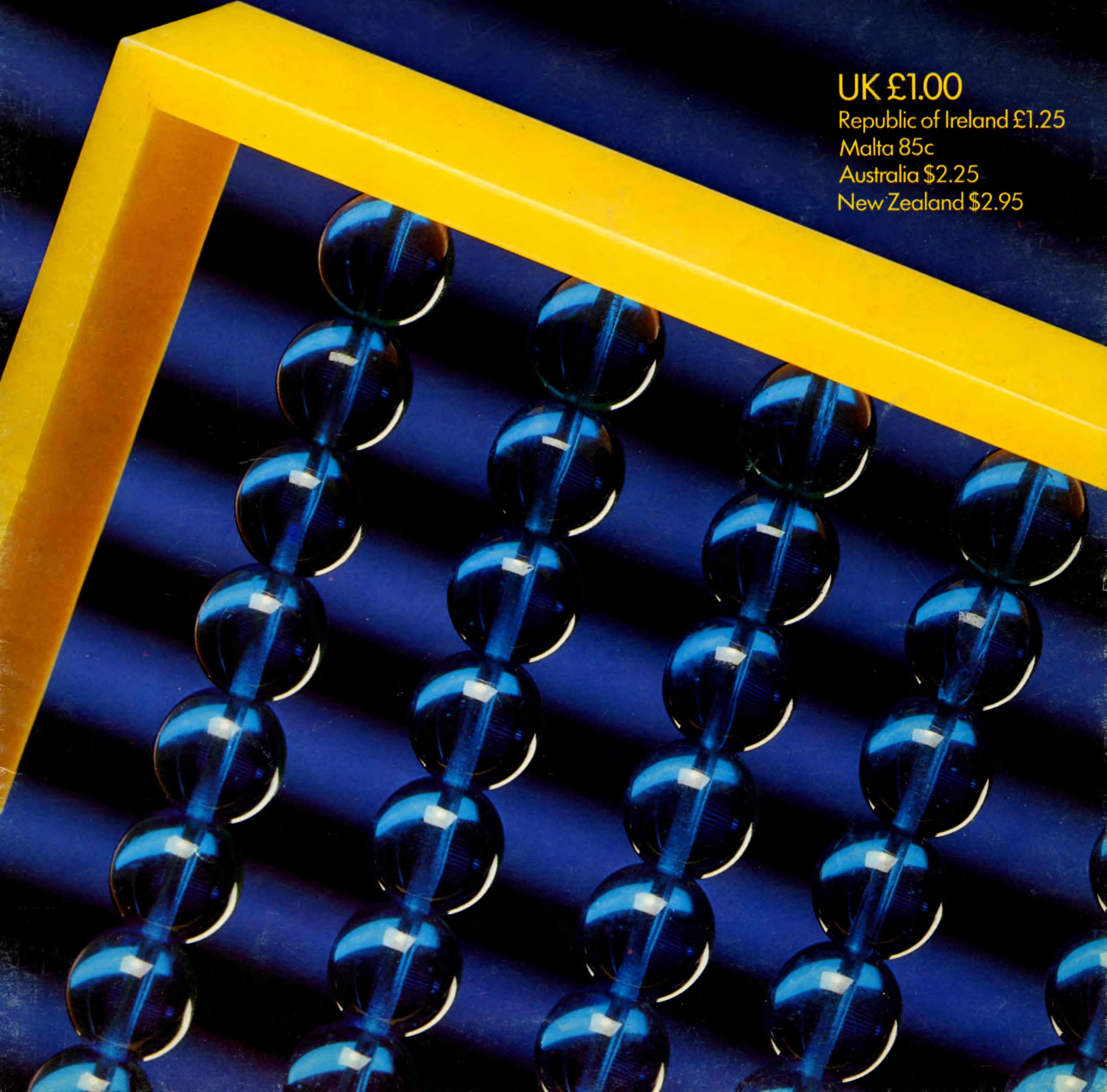
A MARSHALL CAVENDISH 4 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE

UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95



INPUT

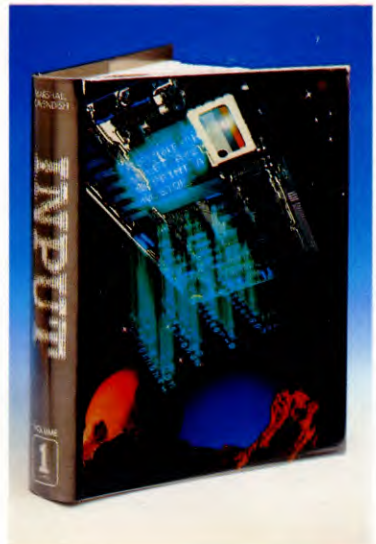
Vol 1 **No 4**

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front Cover, Dave King. Pages 129, 130, 134, Peter Bentley. Pages 136, 138, 140, Graeme Harris. Page 142, Graeme Harris, David Lloyd. Pages 144, 146, 148, 150, Ian Craig. Pages 152, 154, Dick Ward. Page 156, Dave King. Pages 158, 160, Dave King.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland: Send £4.95 (inc p & p) (IR£5.45) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Gordon and Gotch Ltd, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Copies of any part of INPUT can be obtained from the following addresses at the regular cover price, with no extra charge for postage and packing:

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services, Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent

COPIES BY POST

Our Subscription Department can supply your copies direct to you regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. These rates apply anywhere in the world. Send your order, with payment to:
Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN
Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries—and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

SCORING AND TIMING ROUTINES

- MINEFIELD GAME
- KEEPING SCORE
- ADDING A GAME TIMER
- QUICK DRAW REACTION TIMER

There's nothing like knowing that you only have two seconds to get back to base or that you're within ten points of the high score to get your pulse racing. All arcade games keep some sort of count of the score or time to add to the excitement—and with a few simple programs, so can you.



Nearly all computer games need some kind of scoring and timing—or even both. Without them you can't judge how good you are at the game, or if you're improving, and there's little point in playing against your friends.

You could have someone sitting beside you counting your hits on the ferocious foe, but there's little point when you can program the computer to do it for you. A few more program lines will persuade the machine to remember high scores, too.

Similarly, there's no need to resort to a stopwatch for timing. As you saw in the maze game (pages 68 to 74), the machines all have a built-in timer, and it can be used in many ways to improve your games.

MINEFIELD

So that you can see how to set up scoring and timing routines in practice, here is a game, suitable for all the machines except the ZX81 and Vic 2Ø, which will have the routines added to it progressively. Each routine is quite simple and could be added to other games, too.

The game is called Minefield and in it you are a tank commander whose mission is to rescue paratroopers who are recklessly landing in a minefield. Wherever the tank is moved it runs the risk of detonating a mine planted at random by the computer. As in a real minefield, the mines are invisible—so move with caution!

The tank (only a # mark, unfortunately, until you learn how to combine graphics and movement in a BASIC program) is controlled by using the usual movement keys—Z for left, X for right, P for up and L for down. Indeed, the core of the program is the 'moving around the screen routine' which you saw earlier (pages 54 to 59).

When you type in and RUN this section of the game you'll see that it's not yet complete—once you've rescued the paratrooper nothing happens except that the tank continues to wander around the screen aimlessly. The program has to be stopped by pressing the BREAK, ESCAPE or RUN/STOP keys, or you'll have to wait until the tank detonates a hidden mine. But don't worry, this will be improved as soon as you add the scoring and timing routines which follow.

On the Tandy, use 247 instead of 223 in Line 14Ø and 15Ø; 251 instead of 239 in Line 16Ø; 253 instead of 247 in Line 17Ø.

```
5Ø LET PO=21Ø
6Ø CLS
7Ø PRINT@ 256, STRING$ (32,"—")
9Ø LET X=RND(256)-1
11Ø IF X< >PO THEN PRINT@ X,"O";
   ELSE GOTO 9Ø
12Ø PRINT@ PO,"#";
13Ø LET LP=PO
14Ø IF PEEK(34Ø)=223 THEN
   LET PO=PO-1:GOTO 19Ø
15Ø IF PEEK(338)=223 THEN
   LET PO=PO+1:GOTO 21Ø
16Ø IF PEEK(338)=239 THEN
   LET PO=PO-32:GOTO 22Ø
17Ø IF PEEK(342)=247 THEN
   LET PO=PO+32:GOTO 22Ø
18Ø GOTO 14Ø
19Ø IF (LP AND 31)=Ø THEN
   LET PO=LP
2ØØ GOTO 22Ø
21Ø IF (PO AND 31)=Ø THEN
   LET PO=LP
22Ø IF PO>255 OR PO<Ø THEN
   LET PO=LP:GOTO 14Ø
23Ø PRINT@ LP,"□";
24Ø PRINT@ PO,"#";
25Ø LET M=RND(256)-1
27Ø IF M=PO THEN PRINT@ PO,"□":
   PRINT@ 13Ø,"BOOM!!—YOU'VE HIT
   A MINE":STOP
31Ø GOTO 13Ø
```

The tank is moved around the screen by Lines 14Ø to 17Ø, which check for the appropriate keypresses—see page 59. In order to confine the tank to the top half of the screen, Line 22Ø reads IF PO>255.... Screen location 255 is the last location on the top half of the screen, so the IF ... THEN stops it straying below the dotted line drawn by Line 7Ø.

Line 7Ø draws the line across the screen by using a new function—STRING$. STRING$ (32,"—") as in Line 7Ø simply tells the Dragon to draw a string of 32 dashes. (If you wanted ten question marks you'd use STRING$(1Ø,"?"), and so on.)

The place that the paratroopers drop is chosen by the random number in Line 9Ø, and Line 11Ø displays the parachute on the screen if the position chosen isn't already occupied by the tank. If it is, then Line 9Ø chooses another position.

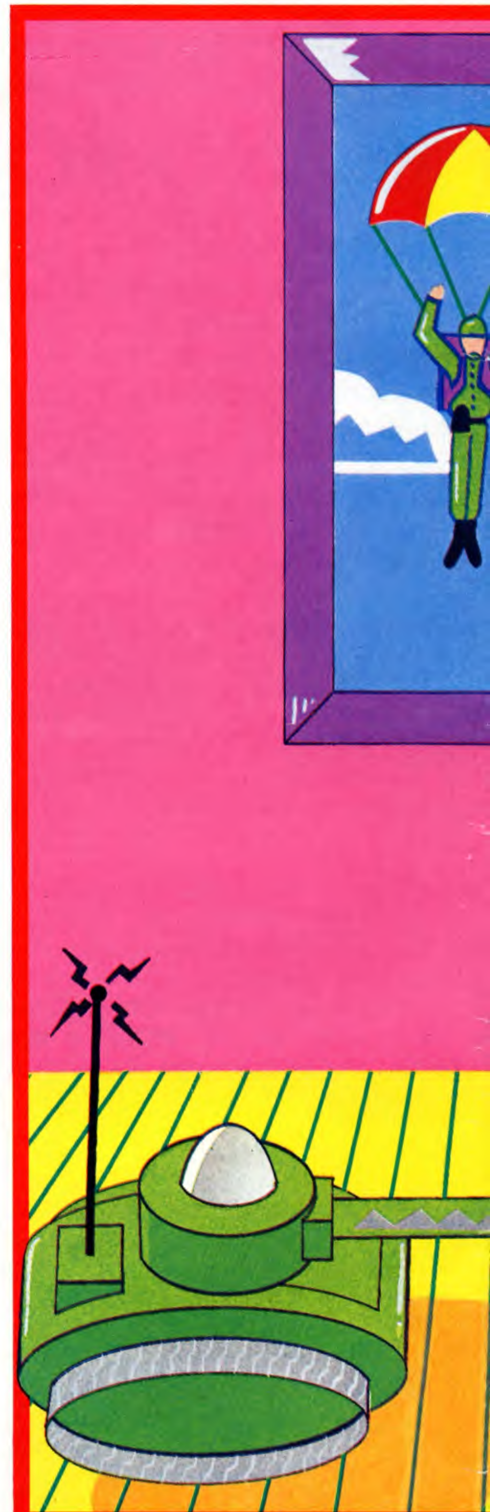
Finally, the position of the hidden mine is chosen by Line 25Ø. Line 27Ø then checks if the tank and mine are at the same screen location. If they are, the explosion message is displayed and the program stops.

```
15 VDU23;82Ø2;Ø;Ø;Ø;
5Ø LET tankx=2Ø : LET tanky=1Ø
6Ø CLS
7Ø PRINT TAB(Ø,13) STRING$ (4Ø,"—")
9Ø LET parax=RND(3Ø)+4
1ØØ LET paray=RND(12)
11Ø IF (parax< >tankx) AND (paray< >
   tanky) THEN PRINT TAB(parax,paray)
   "O";ELSE GOTO 9Ø
12Ø PRINT TAB(tankx,tanky) "#";
13Ø LET lasttankx=tankx : LET
   lasttanky=tanky
14Ø KEY$=GET$
145 IF KEY$="P" THEN tanky=tanky-1
15Ø IF KEY$="L" THEN tanky=tanky+1
16Ø IF KEY$="Z" THEN tankx=tankx-1
17Ø IF KEY$="X" THEN tankx=tankx+1
19Ø IF tankx<5 OR tankx>34 THEN
   LET tankx=lasttankx
2ØØ IF tanky<1 OR tanky>12 THEN
   LET tanky=lasttanky
23Ø PRINT TAB(lasttankx,lasttanky) "□"
24Ø PRINT TAB(tankx,tanky) "#"
25Ø LET minex=RND(3Ø)+4
26Ø LET miney=RND(12)
27Ø IF (minex=tankx) AND (miney=
   tanky) THEN PRINT TAB(minex,miney)
   "□": PRINT TAB(7,7)"BOOM!!—
   YOU'VE HIT A MINE":END
31Ø GOTO13Ø
```

In the Acorn program, Lines 145 to 17Ø check which key has been pressed by the player before moving the tank. A full explanation of how this is done is on page 55. Lines 19Ø and 2ØØ confine the tank to an area above the dotted line drawn by Line 7Ø, and also stop the tank going off the top of the screen.

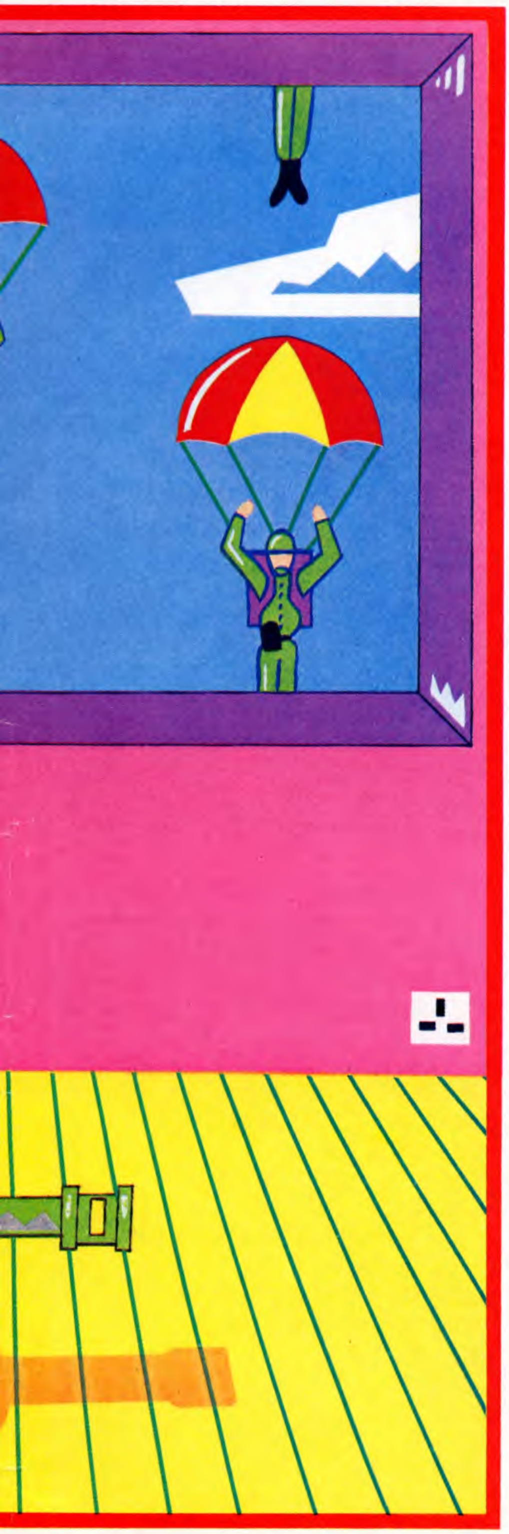
Line 7Ø draws the line across the screen using a new function—STRING$. STRING$ (4Ø,"—") simply tells the Acorn machines to draw a string of 4Ø dashes. (If you wanted a string of 15 plus signs, for example, you'd type STRING$(15,"+").)

The place the paratroopers drop is chosen by Lines 9Ø and 1ØØ. The dropping position



is compared with the tank's position by Line 11Ø. If the positions are different, then the paratrooper is displayed on the screen at the chosen position. If the tank's position is chosen then another dropping position is selected by going back to Line 9Ø.

In a similar way Lines 25Ø and 26Ø choose a place for the mine. Line 27Ø then checks if



the tank and mine are at the same screen position. If they are, the tank is wiped out by PRINTing a space at its position, and the explosion message is also displayed. Then the program stops.

Finally, in order to clear up the display Line 15 is a VDU command which turns off the cursor.

```
50 LET tx=16: LET ty=5
60 CLS
70 PRINT AT 11,0;"- - - - - -
   - - - - - - - - - - - - - - - -
   - - - - - - - - - - -"
90 LET px=INT (RND*30)+1
100 LET py=INT (RND*10)
110 IF px=tx AND py=ty THEN GOTO 90
120 PRINT AT py,px;"O";AT ty,tx;"#"
130 LET txx=tx: LET tyy=ty
140 LET a$=INKEY$
145 IF a$="p" THEN LET ty=ty-1
150 IF a$="l" THEN LET ty=ty+1
160 IF a$="z" THEN LET tx=tx-1
170 IF a$="x" THEN LET tx=tx+1
190 IF ty<0 OR ty>10 THEN LET
   ty=tyy
200 IF tx<1 OR tx>30 THEN LET
   tx=txx
230 PRINT AT tyy,txx;"□"
240 PRINT AT ty,tx;"#"
250 LET mx=INT (RND*30)+1
260 LET my=INT (RND*10)
270 IF mx=tx AND my=ty THEN PRINT
   AT my,mx;"□": PRINT AT 8,3;
   "BOOM!!—YOU'VE HIT A MINE":
   STOP
310 GOTO 130
```

In the Spectrum program the screen is divided in half by Line 70—it PRINTs a series of 32 dashes across the screen. Lines 145 to 170 control the movement of the tank around the screen. A full explanation of how this works is on page 57.

Lines 190 and 200 confine the tank to an area above the dotted line drawn by Line 70, and prevent it from going off the screen.

The dropping position of the paratroopers is chosen randomly by Lines 90 and 100. Line 110 checks if the tank and paratrooper will occupy the same screen location. If they will then a new dropping position is chosen by going to Line 90.

In a similar way Lines 250 and 260 choose a place for the mine. Line 270 compares the mine's position with the tank's. If they are the same then the tank is PRINTed out, and the explosion message is displayed. The program stops if there is an explosion.

```
15 Z$="■■■■■■■■■
   ■■■■":POKE 650,128
50 LET PO=205
60 PRINT "■";Z$;"- - - - -
   - - - - - - - - - - - - - - -
   - - - - - - - - - - - - - - -
   - - - - - - -"
70 POKE 53280,1:POKE 53281,3
85 LET L=5
90 LET X=INT(RND(1)*479)+1
110 IF X=PO THEN GOTO 90
115 POKE 1024+X,87
120 POKE 1024+PO,35
130 LET LP=PO: LET LL=L:GET A$
140 IF A$="Z" THEN LET PO=PO-1:
   L=L-1:GOTO 190
150 IF A$="X" THEN LET PO=PO+1:
   L=L+1:GOTO 210
160 IF A$="P" THEN LET PO=PO-40:
   GOTO 220
170 IF A$="L" THEN LET PO=PO+40:
   GOTO 220
180 GOTO 130
190 IF L<0 THEN LET PO=LP:
   LET L=LL
200 GOTO 220
210 IF L>39 THEN LET PO=LP:
   LET L=LL
220 IF 1024+PO<1024 OR 1024+
   PO>1024+479 THEN LET PO=LP:
   GOTO 130
230 POKE 1024+LP,32
240 POKE 1024+PO,35
250 LET M=INT(RND(1)*479)+1
270 IF M=PO THEN POKE 1024+
   PO,32:PRINT Z$;"■BOOM!!—
   YOU'VE HIT A MINE": STOP
310 GOTO 130
```

In the Commodore program the screen is divided in half by a line drawn by Line 60. Z$ has been defined in Line 15 and is a string of cursor controls. The POKE in Line 15 switches on the key repeat.

The screen and border colours are set by the POKEs in Line 70.

Line 85 sets L, which is used by the program in Lines 190 and 210 to prevent the tank going off either side of the screen.

POKEs are used to display characters on the screen. PO is the position of the tank, and Line 120 displays it on the screen.

The dropping position of the paratrooper is chosen by Line 90. The position is compared with the tank's position. If they are the same then a new position for the paratrooper is chosen. Line 115 displays the paratrooper on the screen.

Line 220 stops the tank going off the top of the screen or below the dotted line. Line 230 blanks out the last position of the tank and Line 240 displays the tank in its new position.

Finally, Line 250 selects at random a position for the mine. Line 270 compares the position of the mine and the position of the tank. If they are the same then the tank is blanked out by POKEing and the explosion message is displayed.

SCORING

In arcade-type games the score is normally increased when the screen positions of two objects are the same. The objects may be a missile and a target, a Pacman and a power pill, a tank and a paratrooper, a horse and a winning post or whatever your game demands.

So add these lines to the program to see how a scoring mechanism works in practice:

```
4Ø LET S=Ø
28Ø IF X=PO THEN LET S=S+1:
   GOTO 9Ø
33Ø PRINT@295,S;"PARATROOPERS";
```

```
4Ø LET S=Ø
28Ø IF (parax=tankx) AND (paray=
   tanky)THEN LET S=S+1:GOTO 9Ø
33Ø PRINT TAB(12,15);S;
   "□PARATROOPERS"
```

```
4Ø LET s=Ø
28Ø IF px=tx AND py=ty THEN LET
   s=s+1:GOTO 9Ø
33Ø PRINT AT 14,8;s;"□PARATROOPERS"
```

```
4Ø LET S=Ø
28Ø IF X=PO THEN LET S=S+1:
   GOTO 9Ø
33Ø POKE5328Ø,254:POKE53281,246:
   PRINT "■□□□□□□□";S;
   "PARATROOPERS"
```

And change the STOP or END in Line 27Ø to GOTO 33Ø. Line 27Ø should now read:

```
27Ø IF M=PO THEN PRINT@ PO,"□":
   PRINT@ 13Ø,"BOOM!!—YOU'VE HIT
   A MINE":GOTO 33Ø
```

```
27Ø IF (minex=tankx) AND (miney=
   tanky) THEN PRINT TAB(minex,miney)
   "!": PRINT TAB(7,7)"BOOM!!—
   YOU'VE HIT A MINE":GOTO 33Ø
```

```
27Ø IF mx=tx AND my=ty THEN
   PRINT AT my,mx;"!": PRINT AT
   8,3;"BOOM!!—YOU'VE HIT A
   MINE": GOTO 33Ø
```

```
27Ø IF M=PO THEN POKE 1Ø24+
   PO,32:PRINT Z$;"■BOOM!!—YOU'VE
   HIT A MINE":GOTO 33Ø
```

Line 28Ø is the important one. It checks if the tank and the paratrooper are on the same screen location. If they are, then the score is increased by 1.

Line 4Ø sets the score to zero before the game starts and Line 33Ø displays the score. Changing the end of Line 27Ø makes the computer display the score after a mine has been detonated.

The player is now given a succession of paratroopers to rescue. When one is rescued another drops from the sky. The game stops when a mine is detonated by the tank moving across the same screen location.

HIGH SCORES

Adding a high score feature to a game isn't difficult. All you need is a variable for the high score—say, HS—and some way of updating it as the high score is exceeded, plus a display routine.

These are the lines you'll have to add to get a high score feature:

```
3Ø LET HS=Ø
35Ø IF S>HS THEN LET HS=S
37Ø PRINT@ 424,"HIGH SCORE=";HS;
```

```
3Ø LET HS=Ø
35Ø IF S>HS THEN LET HS=S
37Ø PRINT TAB(12,2Ø) "HIGH
   SCORE=";HS
```

```
3Ø LET hs=Ø
35Ø IF s>hs THEN LET hs=s
37Ø PRINT AT 18,8;"HIGH
   SCORE=";hs
```

```
3Ø LET HS=Ø
35Ø IF S>HS THEN LET HS=S
37Ø PRINT "■□□□□□□□□□
   HIGH SCORE=";HS
```

First you must set the high score to the lowest level possible, so Line 3Ø sets HS at zero. After the game has stopped Line 35Ø compares the latest score (S) with the high score (HS). If the latest score is higher than the high score, then HS is updated so that it is the same as S. Finally, Line 37Ø displays the high score on the screen.

These lines will probably seem to be enough to add a high score to the game. Unfortunately this isn't true. Every time you tell the computer to RUN the program it automatically forgets the value of HS, along with the values of all the other variables. To keep the value of HS you'll need to add the 'Do you want another go?' lines that have been used previously (page 35):



```
39Ø FOR F=1 TO 1ØØØ:NEXT F
4ØØ PRINT@ 13Ø
41Ø PRINT@ 135,"ANOTHER GO?
   (Y/N)";
42Ø LET K$=INKEY$:IF K$="" THEN
   GOTO 42Ø
43Ø IF K$="Y" THEN GOTO 4Ø
44Ø IF K$="N" THEN END
45Ø GOTO 42Ø
```

```
39Ø FOR F=1 TO 4ØØØ: NEXT F
4ØØ PRINT TAB(7,7) STRING$(26,"□")
41Ø PRINT TAB(11,8)"ANOTHER GO? (Y/N)"
42Ø LET K$=GET$
43Ø IF K$="Y" THEN GOTO 4Ø
44Ø IF K$="N" THEN END
45Ø GOTO 42Ø
```

```
39Ø PAUSE 1ØØ
4ØØ PRINT AT 8,3; TAB31
41Ø PRINT AT 8,7;"ANOTHER GO? (Y/N)"
```



```
420 LET a$=INKEY$
430 IF a$="y" THEN GOTO 40
440 IF a$="n" THEN STOP
450 GOTO 420
```

```
390 FOR F=1 TO 1000:NEXT F
410 PRINT "■□□□□□□□□□
    ANOTHER GO? (Y/N)";
420 GET K$:IF K$="" THEN GOTO 420
430 IF K$="Y"THEN GOTO 40
440 IF K$="N" THEN PRINT "■":
    POKE 650,0:END
450 GOTO 420
```

This is what the extra lines do:

There is a short delay introduced by the FOR ... NEXT loop in Line 390 on the Acorn, Commodore, Dragon and Tandy and by the PAUSE in the Spectrum program. Line 400 in the Acorn, Dragon, Tandy and Spectrum programs clears a portion of the screen ready for the message ANOTHER GO? (Y/N) to be PRINTed by Line 410.

The 'Another Go?' routine is on Lines 410 to 450—Line 430 restarts the program at Line 40 if Y is pressed, and Line 440 stops the program if N is pressed. Line 450 makes sure that any other keypress is ignored.

As there is no need now to type RUN every time you wish to have another play the value of HS is preserved, although reLOADing the program will lose the value of HS even if you find some way of starting the program without typing RUN.

TIMING

As it stands the game depends too much on luck—the player just keeps on going until he runs over the concealed mine.

Some skill can be introduced into this kind of game by turning it into a race against the clock. By making these additions you can time how long it takes to rescue ten paratroopers.

```
80 TIMER=0
290 IF S<10 AND X=PO THEN GOTO 90
300 IF S=10 THEN GOTO 320
320 LET T=TIMER
340 IF S=10 THEN PRINT@ 327,"IN";
    T/50;"SECONDS"
```

```
10 LET T=0
80 TIME=0
290 IF S<10 AND (parax=tankx) AND
       (paray=tanky) THEN GOTO 90
300 IF S=10 THEN GOTO 320
320 LET T=TIME
340 IF S=10 THEN PRINT TAB(12,16)
    "IN□";T/100;"□SECONDS"
```

```
10 LET T=0
80 POKE 23672,0: POKE 23673,0
290 IF s<10 AND px=tx AND py=ty
    THEN GOTO 90
300 IF s=10 THEN GOTO 320
320 LET t=PEEK 23672+256*PEEK
    23673
340 IF s=10 THEN PRINT AT 16,8;
    "IN□";t/50;"□SECONDS"
```

```
80 LET TI$="000000":L=5
290 IF S<10 AND X=PO THEN GOTO 90
300 IF S=10 THEN GOTO 320
320 LET T=VAL(RIGHT$(TI$,2))+
    60*VAL(MID$(TI$,3,2))
340 IF S=10 THENPRINT
    "■□□□□□□□□□IN";T;
    "□SECONDS"
```

The timer in each of the machines is running all the time the computer is switched on. All you have to do to 'start' the timer, then, is to set the timer reading to zero. The line which resets the timer is Line 80—you'll notice that to reset the Acorn timer you enter TIME=0, on the Commodore LET TI$="000000", on the Dragon/Tandy TIMER=0 and on the Spectrum POKE 23672,0: POKE 23673,0. What the Acorn, Commodore, Dragon and Tandy lines are doing is clear, but Spectrum owners may be a little puzzled. The two POKEs set the values of two particular memory locations to zero. What happens is that the first memory location counts up to 255 before overflowing into the second.

The clock is 'stopped' by Line 320. The clock cannot actually be stopped, so you get the machine to remember what the reading was at a particular instant—when two objects coincide on the screen, say. In these programs the timer reading is called T— the Acorn says LET T=TIME, the Commodore LET T=VAL(RIGHT$(TI$,2) + 60*VAL(MID$(TI$,3,2)), the Dragon/Tandy LET T=TIMER and the Spectrum LET t=PEEK 23672+256*PEEK 23673. Again, the Spectrum method may be a little puzzling. It's looking directly at the values of the two memory locations which were set to zero earlier in the program. Location 23672, like any memory location in an 8-bit micro, can contain whole numbers between 0 and 255. Location 23673 goes up 1 every time location 23672 overflows.

So to work out the total amount of time that the Spectrum timer has been running you have to multiply the value in location 23673 by 256 and add on the value stored in location 23672.

You may now see why t is equal to PEEK



23672 + 256*PEEK 23673. (In any case PEEK and POKE will be dealt with fully in a later article.)

The Commodore clock works by updating a six character string as set up by Line 8Ø. Starting from the right the string contains two digits for hours, then two for minutes and then two for seconds. Line 32Ø evaluates the string to get a reading in seconds. The number of minutes that have elapsed are multiplied by 6Ø and added to the number of seconds—MID$(TI$,3,2) looks at the number of minutes, and RIGHT$(TI$,2) looks at the number of seconds. (Again, a full explanation of string slicing comes in a later article.)

The clock must be stopped when the player has rescued ten paratroopers, so Line 3ØØ checks if ten paratroopers have been rescued then jumps to Line 32Ø which 'stops' the timer. In the Commodore program Line 34Ø PRINTs the time when ten paratroopers have been rescued.

If a paratrooper has been successfully rescued *and* the total number rescued is less than ten, Line 29Ø makes another drop down.

Line 34Ø displays the rescue time *only* if ten have been recovered. The timer reading is divided by 5Ø in the Dragon, Tandy and Spectrum programs, and by 1ØØ in the Acorn program. Thus the time displayed is in seconds—the Acorn clock is updated 1ØØ times a second and the Dragon and Spectrum clocks are updated 5Ø times a second.

LOW TIME

Just as a high score was added to the game earlier, you might like to add a low time feature. In this particular game you'll be able to record the fastest time in which the ten paratroopers have been recovered, although the principle can be applied to any timing you might wish to do.

These are the lines to add:

```
20 LET LT=999999
360 IF T<LT AND S=10 THEN LET
   LT=T
380 PRINT@ 452,"LOW TIME=";LT/50;
   "SECONDS"
```

```
20 LET LT=999999
360 IF T<LT AND S=10 THEN LT=T
380 PRINT TAB(12,21) "LOW TIME=";
   LT/100;"□SECONDS"
```

```
20 LET lt=999999
360 IF t<lt AND s=10 THEN LET
```

```
   lt=t
380 PRINT AT 21,4;"LOW TIME=";
   lt/5Ø;"□SECONDS"
```

```
2Ø LET LT=999999
36Ø IF T<LT AND S=1Ø THEN LET
   LT=T
38Ø PRINT"■□□□□□□□□□
   LOW TIME=";LT;"□SECONDS"
```

Just as when the high score was set up a very low 'high score' was originally set, now a ridiculously long 'low time' is set. Line 2Ø sets the low time variable (LT or lt) to a value of 999999.

Line 36Ø goes on to compare the latest time with the low time. If the latest time is shorter than the low time *and* ten paratroopers have been rescued, then the low time is altered so that it is the same as the latest time.

Finally, Line 38Ø displays the low time in seconds. The low time variable is divided by 5Ø or 1ØØ to display the low time in seconds.

One thing you must remember is that if you are using a low time feature in a game you need to use the 'Another go?' routine otherwise you'll lose the value of the low time every time you RUN the program.

TIMING AND THE KEYBOARD

So far you've seen how to control the machine's timer from within a program by checking the positions of two objects on the screen. Another way to 'stop' the clock is to use the keyboard.

On page 54 you were shown how to use INKEY$ or GET$. They can be used just as easily to start or stop the timer as to control objects on the screen.

Here is a 'quick-on-the-draw' game which illustrates how the keyboard can be used to stop the timer:

```
2Ø CLS
3Ø LET N=RND(9ØØ)
4Ø FOR F=Ø TO N
5Ø NEXT F
6Ø PRINT@ 269,"DRAW!!"
7Ø TIMER=Ø
8Ø IF INKEY$="" THEN GOTO 8Ø
9Ø LET T=TIMER
1ØØ PRINT@ 269,"BANG!!"
11Ø FOR F=1 TO 3ØØ
12Ø NEXT F
13Ø LET M=RND(25)
14Ø IF T<M THEN PRINT@ 264,
   "YOU'VE SURVIVED"
15Ø IF T>M THEN PRINT@ 266,
   "YOU'RE DEAD"
16Ø IF T=M THEN PRINT@ 264,
   "YOU'RE BOTH DEAD"
```

```
2Ø CLS
3Ø LET N=RND(2ØØØ)
4Ø FOR delay=Ø TO N
5Ø NEXT delay
6Ø PRINT TAB(17,13) "DRAW!!"
7Ø TIME=Ø
8Ø K$=GET$
9Ø LET T=TIME
1ØØ PRINT TAB(17,13) "BANG!!"
11Ø FOR delay=1 TO 1ØØØ
12Ø NEXT delay
13Ø LET M=RND(4Ø)
14Ø IF T<M THEN PRINT TAB(12,13)
   "YOU'VE SURVIVED"
15Ø IF T>M THEN PRINT TAB(14,13)
   "YOU'RE DEAD"
16Ø IF T=M THEN PRINT TAB(12,13)
   "YOU'RE BOTH DEAD"
```

```
2Ø CLS
3Ø LET n=INT (RND*4ØØ)+1
4Ø PAUSE n
6Ø PRINT AT 11,14;"DRAW!!"
7Ø POKE 23672,Ø: POKE 23673,Ø
8Ø IF INKEY$="" THEN GOTO 8Ø
9Ø LET T=PEEK 23672+256*PEEK
   23673
1ØØ PRINT AT 11,14;"BANG!!"
11Ø PAUSE 5Ø
13Ø LET m=INT (RND*35)+1
14Ø IF t<m THEN PRINT AT 11,9;
   "YOU'VE SURVIVED"
15Ø IF t>m THEN PRINT AT 11,11;
   "YOU'RE DEAD"
16Ø IF t=m THEN PRINT AT 11,9;
   "YOU'RE BOTH DEAD"
```

Q+A

Is there any limit to the period I can time?

Yes, there is—though it is usually so long that in practice it makes no difference. The timer in most home computers runs at a similar speed, and the limiting factor is how many time pulses the computer is able to remember. The Dragon, Tandy and Commodore can hold up to two bytes (65,535) which works out to about 22 minutes. The Spectrum runs to three bytes—nearly four days. The BBC and the Electron store the time in four bytes which means they can be set to count for almost 350 years!

```
2Ø PRINT "■"
3Ø LET N=INT(RND(1)*9ØØ)+1
4Ø FOR F=Ø TO N
5Ø NEXT F
6Ø PRINT "■■■■■■■";TAB(16);
   "DRAW!!"
7Ø LET TI$="ØØØØØØ":POKE 198,Ø
8Ø GET K$:IF K$=""THEN GOTO 8Ø
9Ø LET T=TI
1ØØ PRINT "■■■■■■■";TAB(16);
   "BANG!!"
11Ø FOR F=1 TO 3ØØ
12Ø NEXT F
13Ø LET M=INT(RND(1)*35)+1
14Ø IF T<M THEN PRINT
   "■■■■■■■■■" TAB(51)
   "YOU'VE SURVIVED"
15Ø IF T>M THEN PRINT
   "■■■■■■■■■" TAB(53)
   "YOU'RE DEAD"
16Ø IF T=M THEN PRINT
   "■■■■■■■■■" TAB(5Ø)
   "YOU'RE BOTH DEAD"
```

The program displays 'DRAW!!'—the player then has to press any key as quickly as possible. The reaction is timed from the moment of display and the player is told whether the computer or he has won.

The program is very simple. After Line 2Ø has cleared the screen a random pause is introduced by Lines 3Ø to 5Ø. Line 6Ø displays 'DRAW!!' and the timer is started immediately by Line 7Ø. Line 8Ø makes the machine wait until a key is pressed before continuing. The line is exactly as was used in 'Keyboard Control' (see page 54).

Immediately any key has been pressed, the program continues to Line 9Ø which effectively stops the timer by calling the timer reading T. Line 1ØØ displays 'BANG!!'.

There is a pause introduced by Lines 11Ø and 12Ø (Line 11Ø only on the Spectrum) before a draw time is chosen for the machine—Line 13Ø does this.

Now the machine has two variables, your time T, and the machine's 'time' M. Lines 14Ø to 16Ø compare them and display whatever the outcome of the gunfight was.

If you are using an Acorn machine, don't be tempted to use INKEY to cause the delays at Lines 3Ø and 11Ø as you did in the earlier programs on animation. The problem now is that the delay is cut short if you press a key. So if you happen to keep your finger on a key, the computer will flash through this game in a fraction of a second.

ALL ABOUT READ AND DATA

Getting the computer to READ a store of DATA saves you typing in long programs which are slow to RUN. You can use the technique for anything from graphics to simple indexes.



One of the features that makes a computer so versatile is the use of variables. And for the most part, assigning values to them is fairly straightforward—you can simply say LET X=5, for example. But sometimes the amount of such information that you want to use can get out of hand. That is where DATA statements, and the accompanying READ and RESTORE commands, come in handy. Note that this is not available in ZX81 BASIC.

The word DATA has a very specific meaning here. 'Data' tends to be used to mean a variety of things. For example, a program, a machine code routine and an array stored on cassette tape or floppy disk are all called 'data'. But in this article it just means the DATA statement, and the items stored in it.

The first method of putting something into a variable that people come across—whether it is giving a value to a numeric variable, or entering a string—is through a simple INPUT statement. But that is only of use when information is being supplied to the computer by the user, each time the program is RUN.

Often, however, there are fixed values which do not need to be entered or altered by the operator, and so can be permanently incorporated into the program. This is, of course, where the LET statement comes in. But if there is a lot of information, using masses of LETs is tedious to program and slow to RUN.

Here's an example where a fixed list of headings might be used in a program for printing out a document:

```
10 LET A$ = "DAY"
20 LET B$ = "WEEK"
30 LET C$ = "MONTH"
40 LET D$ = "YEAR"
50 PRINT A$,B$,C$,D$
```

However, you can also use DATA statements to set them up instead.

```
10 READ A$,B$,C$,D$
20 PRINT A$,B$,C$,D$
100 DATA DAY,WEEK,MONTH,YEAR
```

If you now wanted to use not four but fifty different headings, the first program would require an extra 46 statements, the second only one or two, depending on the computer and the size of the headings.

Note that on the Spectrum the DATA words must be surrounded by quotation marks. So Line 100 is written as:—

```
100 DATA "DAY","WEEK","MONTH",
    "YEAR"
```

HOW READ AND DATA WORK

How, in fact, do the READ and DATA statements work? When the computer comes across a READ instruction, it scans through the program until it comes across the first DATA statement. Then it assigns the first item of DATA to the variable in the READ command.

So in the program above, the DATA string "DAY" is assigned to the variable A$, then "WEEK" is assigned to B$ and so on.

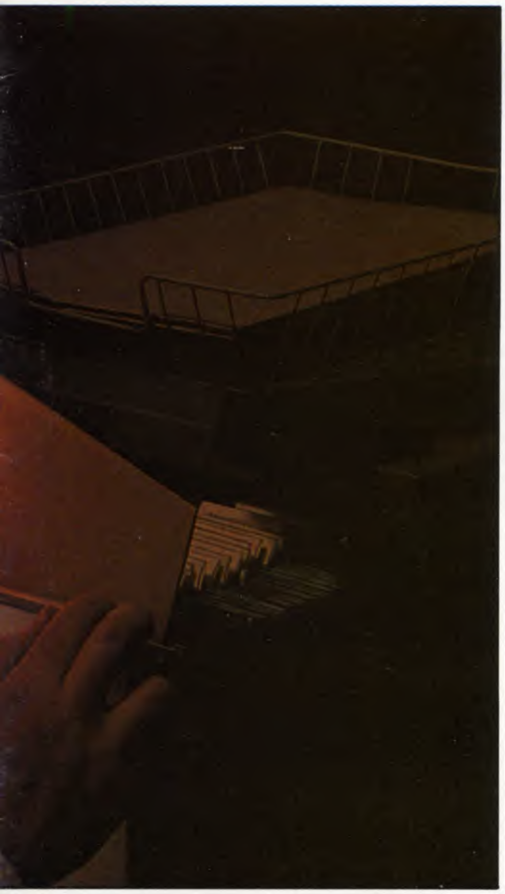
DATA statements usually appear at the end of a program, where they do not 'get in the way' while the rest of the program is being written and de-bugged. But in fact they can go anywhere at all. The computer simply ignores any DATA statement unless a READ statement makes it do otherwise. This next program would work perfectly well:

```
10 DATA GERMANY
20 READ A$,B$
30 DATA FRANCE,ITALY,SPAIN
40 READ C$,D$
```

Obviously, though, the program is easier for you to read if all the DATA is gathered together in one place. And there is one firm rule: the DATA *must* appear in the order that the computer will READ it.

The number of items a single DATA list can hold depends on the maximum line length of your computer. As soon as one line is full just start another—the computer still takes them in order.

- HOW **READ** AND **DATA** WORK
- DIFFERENT TYPES OF **DATA**
- AVOIDING PROBLEMS
- USING **DATA** TO MAKE A SIMPLE DIRECTORY
- THE **RESTORE** STATEMENT
- USES FOR **DATA** LISTS
- SELECTING FROM **DATA**
- HOW TO DRAW SIMPLE GRAPHICS FROM **DATA**



DIFFERENT TYPES OF DATA

So far the DATA statements have been strings, but they could just as well be numbers. The Spectrum and Acorn computers allow you to use variables and functions as well, so you could have a line looking something like this:

```
DATA PARIS, 256,a*5
```

(Again, you would need quotation marks around the "PARIS" on the Spectrum.)

The variables in the READ command must be placed in the same order as the relevant DATA items. So the DATA line above would be read with the command:

```
READ A$,N,X
```

Note that the first variable is a *string* variable to match the first DATA item. But the next two are *numeric* variables because they are READing numbers.

PROBLEMS WITH DATA

It is easy to make mistakes when entering DATA. The main problems occur when you do not have enough items or you try to READ in the wrong kind. If the DATA line above had been wrongly typed in as DATA 256,PARIS,a*5 then A$ would be '256'. Although wrong, except on the Spectrum it would not be rejected.

It would be a different matter when the computer tried to READ the string 'PARIS' into the variable N. One of two things would normally happen. The computer might notice that the DATA was the wrong type and come back with an error message such as 'type mismatch' or 'bad data'. Alternatively, the computer might assume that PARIS was a variable name which it had not previously come across, and so would reply with something like 'variable not found'.

The other common error is supplying too few DATA entries. This often happens when you are using a loop containing the READ command, as in the program below. A slight mistake in the loop parameter, or an item of DATA accidentally left out, and the computer will try to READ past the end of the DATA list. The result is that the program stops with an error message like 'out of data'.

SIMPLE DIRECTORY

Here is a program that uses a loop to READ the DATA. It is a very simple telephone directory program. You enter the name of the person and the computer PRINTs out the number.

```
5 CLS
10 PRINT TAB(10,2) "TELEPHONE
   DIRECTORY"
20 INPUT TAB(5,10) "PLEASE ENTER
   THE NAME□",R$
30 FOR J=1 TO 5
40 READ N$,T$
50 IF N$=R$ THEN PRINT TAB(5,12)
   N$;"'S NUMBER IS:□";T$:END
60 IFN$="END" THEN PRINT TAB(5,12)
   R$;"□IS NOT IN THE LIST"
70 NEXT J
500 DATA JENNY,62-4325,PETER,499-
   6262,DEBBIE,567-02206,MARK,44-
   9976,END,END
```

```
5 CLS
10 PRINT@ 71,"TELEPHONE DIRECTORY"
12 PRINT
20 INPUT "PLEASE ENTER NAME";R$
30 FOR J=1 TO 5
40 READ N$,T$
50 IF N$=R$ THEN PRINT@ 320,N$"'S
   NUMBER IS:□";T$:END
60 IF N$="END" THEN PRINT@ 320,R$;
   "□IS NOT IN THE LIST"
70 NEXT J
500 DATA ETHEL,38-42765,HUBERT,
   997-403,ARCHIBALD,77-40628,
   GERTRUDE,5666-99994,END,END
```

```
5 PRINT "♥"
10 PRINT TAB(10)"TELEPHONE
   DIRECTORY"
20 INPUT "■PLEASE ENTER THE
   NAME";R$
25 PRINT
30 FOR J=1 TO 5
40 READ N$,T$
50 IF N$=R$ THEN PRINT N$;"'S NUMBER
   IS:□";T$:END
60 IF N$="END" THEN PRINT R$;"□IS
   NOT IN THE LIST"
70 NEXT J
500 DATA TOM,21-23233,JOHN,01-32358,
   JOE,22-45785,JANE,78-87779,END,
   END
```

```
5 CLS
10 PRINT AT 2,6;"TELEPHONE
   DIRECTORY"
20 INPUT "PLEASE ENTER THE NAME",
   R$
30 FOR J=1 TO 5
40 READ N$,T$
50 IF N$=R$ THEN PRINT AT 10,1;N$;
   "'S NUMBER IS:□";T$: STOP
60 IF N$="END" THEN PRINT AT
   10,3;R$;"□ IS NOT IN THE LIST"
70 NEXT J
500 DATA "JENNY","43-15263",
   "DICK","43-14123","JULIE",
   "13-65312","BUSTER","13-71824",
   "END","END"
```

Type in the program, but use the names and telephone numbers of your own friends in the DATA line. You can put in as many DATA items as you like but be sure to adjust the loop counter in Line 3Ø.

Note that the DATA list is terminated with END,END. This is so Line 6Ø can check if the end of the list has been reached. If the program has read this far it means it hasn't found the name and it PRINTs out a message to tell you so. END (or "END" on the Spectrum) has to be entered twice because Line 4Ø reads in *two* DATA items at a time and you would get an 'out of data' error otherwise.

Also note that the telephone number is read as a string. This is because it contains a punctuation mark—the dash—and so cannot possibly be a number. Strings in DATA can contain spaces if you like but not commas. Since DATA items are separated by commas, the computer would treat the comma as the end of that item. If you do need to enter a DATA string that contains a comma then enclose the whole string in quotation marks. This would be necessary in a DATA line like this:

```
1Ø READ N$,A$,B$,C$
2Ø DATA ALBERT SMITH,"23, HIGH
   STREET", BRIGHTON,BT6 4ST
```

The first line of the address has to be written in quotes because of the comma after the number 23. (You would have to do this on the Spectrum anyway, for all these strings.)



USING RESTORE

With the methods covered so far, a program can READ through a list of DATA only once, unless you reRUN the program. To reREAD DATA you must incorporate a RESTORE statement within your program.

For example, say you wanted to look up the numbers of several of your friends. The obvious thing to do is to put in an 'Another go' routine. First replace END or STOP in Line 5Ø with GOTO 8Ø then add the following program lines:

```
8Ø PRINT TAB(2,2Ø)"DO YOU WANT
   ANOTHER NUMBER (Y/N)?"
9Ø K$=GET$
1ØØ IF K$="Y" THEN GOTO 5
11Ø END
```

```
8Ø PRINT AT 12,Ø;"DO YOU WANT
   ANOTHER NUMBER? Y/N"
9Ø PAUSE Ø
1ØØ IF INKEY$="y" THEN GOTO 5
11Ø IF INKEY$="n" THEN GOTO 2ØØØ
```

```
8Ø PRINT @32,"DO YOU WANT ANOTHER
   NUMBER(Y/N)?"
9Ø K$=INKEY$:IF K$="" 
   THEN GOTO 9Ø
1ØØ IF K$="Y" THEN GOTO 5
11Ø END
```

```
8Ø PRINT TAB(4)"DO YOU WANT
   ANOTHER NUMBER(Y/N)?"
9Ø GET K$:IF K$="" THEN GOTO 9Ø
1ØØ IF K$="Y" THEN GOTO 5
11Ø END
```

But what happens when you try to RUN the program? The first time through, the program works fine. But when you press a key for another go it fails due to lack of further DATA. This is because it came to the end of the DATA list on the first RUN. Fortunately there is an easy solution to this problem. Add:

```
15 RESTORE
```

This time the program keeps working every time. This is because the RESTORE command instructs the computer to go right back to the beginning of the DATA list.

When you are developing a program it is a good idea to put a RESTORE statement near the beginning. This stops you from running out of DATA on trial RUNs. If you are likely to want to take out the RESTORE line later, put a REMark statement after it to remind you.

By using RESTORE you can reuse a DATA list as often as necessary. This is particularly

Microtip

Keeping track of your DATA

Whatever they refer to, DATA lines often have one thing in common—it takes a good deal of time to work them out and type them in. This is all very well the first time round, but you may well need to come back to a program much later.

Arranging the DATA neatly only takes a little longer, and can save you ages in the end. The best plan to follow depends on the DATA itself.

Where the DATA defines a block graphic or something similar, arrange the program lines to correspond directly to the graphic rows. If the DATA follows a repetitive format (such as entries in a telephone directory, for example), arrange each entry in exactly the same way in the program lines.



useful in programs where things like labels, or headings for tables, need to be repeated at different times. For example, you can put the months of the year in a DATA list for some sort of calendar program which makes repeated use of these in tables or graphs.

The RESTORE instruction is particularly useful when you want to keep READing through a list to search for a particular item—as the telephone directory program showed.

USES FOR DATA

DATA lists are extremely useful in all sorts of programs. You may have already used them for the user defined graphics (pages 8 to 15) and for drawing a maze (pages 68 to 74). You could also have the computer reading parameters for music or sound effects.

In adventure games, large numbers of DATA lines are frequently used to contain all the necessary text. Quizzes can store all the questions and answers in DATA lines. And arcade games in BASIC often use them to define both characters and backgrounds.

Experienced programmers use DATA for assembly language or machine code programs. Such a program may consist of a short loop containing POKE commands which READ from DATA lists.

In short, any program which contains standard texts, figures or functions can make good use of DATA lists. And with careful use, these lists can become a powerful programming tool.

USING RESTORE POINTERS

The only problem with DATA so far is that the information must always be called in the same sequence, starting from the same place. Using RESTORE, you can always go back to the beginning of a list even if you have not reached the end. But how do you jump into the middle of a list?

On Acorn computers, and the Spectrum, there is a way round this. Rather than using just one list, you can actually use several, directing the computer to the appropriate one. In the graphics program (right and overleaf) try adding these extra lines:

```
15 INPUT'' "<T>HIN OR <F>AT
   HOUSE ",H$
16 IF H$="T" THEN RESTORE 2000
17 IF H$="F" THEN RESTORE 1000
18 IF H$<>"T" AND H$<>"F" THEN
   GOTO 15
2000 DATA 400,100,900,100,900,600,
   400,600,400,100
2010 DATA 500,200,550,200,550,400,
   500,400,500,200,750,200,800,200
   800,400,750,400,750,200
2020 DATA 600,100,700,100,700,400
   600,400,600,100
2030 DATA 300,600,1000,600,650,800,
   300,600
```

```
6 INPUT "OLD OR MODERN BRIDGE?";
   y$
7 IF y$="o" THEN RESTORE 1000
8 IF y$="m" THEN RESTORE 2000
9 IF y$<>"o" AND y$<>"m"
   THEN GOTO 5
2000 DATA 83, 84, 85, 86, 87, 88, 89,
   90, 90, 90, 90, 90, 90, 90, 90, 89,
   88, 87, 86, 85, 84, 83
2010 DATA 42, 55, 63, 65, 63, 55, 42
```

The numbers following the RESTORE commands are known as the RESTORE *pointers.* They direct the computer to a particular DATA list beginning at that line number. So in the Acorn program, if you press F for a fat house, the RESTORE pointer becomes 1000, in Line 17. In the Spectrum program, Line 7 has a similar effect.

In fact if you want the computer to go to the very first DATA list a pointer is not necessary. RESTOREing without a line number is the same as RESTOREing with the number of the first DATA line. Line 17 in the Acorn program could have been written as IF H$="F" THEN RESTORE, with precisely the same effect.

Restore pointers are very useful in games programming. In a lander-type game, for example, you can write the program to test whether you have crashed or landed safely. The DATA lists can then contain information for a crash sound or a victory fanfare, and the RESTORE pointer can point to the correct list.

USING DATA FOR GRAPHICS

DATA statements are very useful in graphics programs, to fix the coordinates for plotting and drawing or to call up ROM graphics. Using DATA in this way is ideal for irregular shapes such as those overleaf, as many more program lines would be needed using individual printing or drawing instructions.

In some cases, though, DATA is not much use. And that is when you are plotting very regular shapes where the coordinates or the graphics shape can be calculated. There is an example of this in the Dragon and Tandy program where the castellations on the fort are calculated rather than listed out.

The DATA in the programs below has been split into several lines deliberately. This is because modifying DATA can be difficult unless clear-cut divisions are made—just imagine returning to a program at a later date and trying to match DATA entries to each of the variables. You can, of course, follow through each of the READ instructions, but this is tedious for a long program. So section off each group of DATA statements and—if you can spare the memory—use plenty of REMs to make things a little clearer.

The following program draws a house:

```
10 MODE 0
15 RESTORE
20 FOR N=1 TO 4
30 READ X,Y
40 MOVE X,Y
50 FOR M=1 TO 4
60 READ X,Y
70 DRAW X,Y
80 NEXT M
90 NEXT N
100 READ X,Y
110 MOVE X,Y
120 FOR P=1 TO 3
130 READ X,Y
140 DRAW X,Y
150 NEXT P
160 PRINT'' "PRESS ANY KEY FOR
   ANOTHER GO"
170 Q=GET
180 CLS: GOTO 10
1000 DATA 300,100,1000,100,1000,
   400,300,400,300,100
1010 DATA 400,200,500,200,500,300,
```

```
    4ØØ,3ØØ,4ØØ,2ØØ,8ØØ,2ØØ,9ØØ,2ØØ,
    9ØØ,3ØØ,8ØØ,3ØØ,8ØØ,2ØØ
1Ø2Ø DATA 6ØØ,1ØØ,7ØØ,1ØØ,7ØØ,3ØØ,
    6ØØ,3ØØ,6ØØ,1ØØ
1Ø3Ø DATA 2ØØ,4ØØ,11ØØ,4ØØ,65Ø,6ØØ,
    2ØØ,4ØØ
```

The DATA gives the coordinates for the various parts of the house. Line 1ØØØ gives the main outline, Line 1Ø1Ø the windows, Line 1Ø2Ø the door and Line 1Ø3Ø the roof. The main part of the program just sets up the loops to READ in the relevant bits of DATA and draw the lines. Line 2Ø sets up the loop to draw four rectangles—the main house, the two windows and the door—and Line 5Ø sets the loop to draw four sides to each rectangle. Finally, Lines 12Ø to 15Ø draw the three sides to the roof.

Once the house is drawn you'll be asked if you want another go, and the program will repeat if you press a key—it doesn't matter which one. Notice the RESTORE in Line 15 to allow the program to reREAD the DATA.

This program uses READ ... DATA to help draw a bridge. If you enter and RUN the program in stages it will be easier to see what's happening, and to check that your typing is correct. So:

```
1Ø FOR t=74 TO 8Ø STEP 3
2Ø PLOT 35,t
3Ø DRAW 175,Ø,-2.5
4Ø NEXT t
```

This bit uses a FOR ... NEXT loop to help PLOT three points near the left-hand side of the screen, then draws an arched line from each point. In Line 3Ø, you'll recognise that the 175,Ø means 'to a point 175 pixels to the right of the starting pixel'. The −2.5 makes the line part of a circle, instead of straight.

```
1ØØ FOR n=18 TO 39
11Ø READ a
12Ø PLOT n,45
13Ø DRAW Ø,a
132 PLOT n+188,45
134 DRAW Ø,a
14Ø NEXT n
1ØØØ DATA 7Ø,7Ø,67,67,7Ø,7Ø,6Ø,6Ø,
    57,57,6Ø,6Ø,57,57,6Ø,6Ø,7Ø,7Ø,67,
    67,7Ø,7Ø
```

This is the section that draws the towers. The loop between Lines 1ØØ and 14Ø, plus the number 45 (pixels from the bottom) in Lines 12Ø and 132, PLOTs the bottom points of each of the closely-set vertical lines that draw the towers.

Then Lines 11Ø and 1ØØØ take over. Line 11Ø tells the computer to READ, in Line 1ØØØ, the height (again in pixels) of each of the 22 vertical lines. So the first line is Ø, 7Ø—that is, vertical and 7Ø pixels high—the second line is Ø, 7Ø, the third Ø, 67 and so on. If you want to watch it happening, try inserting a temporary 135 PAUSE 1ØØ after Line 134.

```
3ØØ PLOT Ø,75
31Ø DRAW 255,Ø,-Ø.1
32Ø PLOT Ø,78
33Ø DRAW 255,Ø,-Ø.1
```

These lines draw the roadway, the −Ø.1 producing a slight upward curve. And finally

```
4ØØ FOR r=62 TO 182 STEP 2Ø
41Ø PLOT r,78
42Ø READ b
43Ø DRAW Ø,b
44Ø NEXT r
1Ø1Ø DATA 42,55,63,65,63,55,42
```

These lines produce the vertical ties between the arch and the roadway, the FOR ... NEXT loop helping to PLOT the starting positions of the ties while Lines 42Ø and 1Ø1Ø govern their height.

If you want the program to reRUN automatically, add these lines:

```
5 CLS:RESTORE
45Ø GOTO 5
```

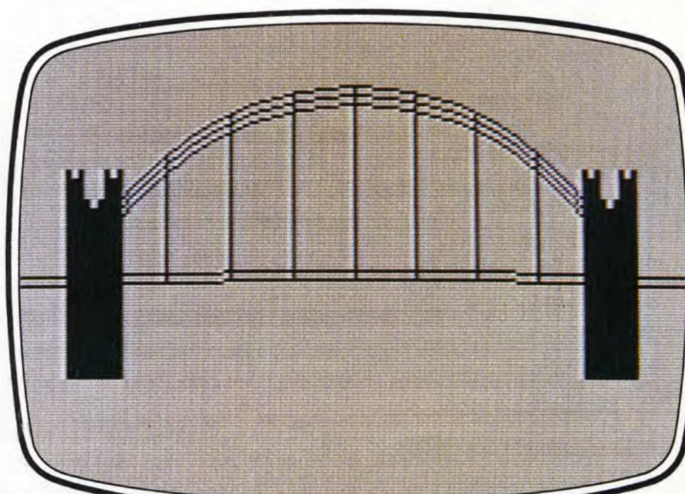
Line 5 clears the screen and then allows the program to reREAD the DATA.

The program here uses a READ ... DATA routine to draw a country cottage from some of the ROM graphics on the Commodore 64:

```
1Ø PRINT"♥"
2Ø POKE 5328Ø,3:POKE 53281,5
3Ø FOR A=1 TO 7
4Ø FOR B=1 TO 14
5Ø READ X
6Ø POKE 1Ø24+291+(A*4Ø)+B,X
65 POKE 55296+291+(A*4Ø)+B,7
7Ø NEXT B
75 NEXT A
5ØØ DATA 32,32,32,32,32,32,32,32,
    32,32,1Ø8,16Ø,123,32
5Ø5 DATA 32,32,233,16Ø,223,32,32,
    32,32,118,16Ø,16Ø,16Ø,116
51Ø DATA 32,233,16Ø,16Ø,16Ø,223,
    32,92,32,16Ø,16Ø,16Ø,16Ø,16Ø
52Ø DATA 32,1Ø2,1Ø2,1Ø2,1Ø2,1Ø2,
    16Ø,16Ø,223,1Ø8,16Ø,1Ø2,16Ø,12Ø
53Ø DATA 32,1Ø2,91,1Ø2,172,1Ø2,
    1Ø2,91,1Ø2,32,32,1Ø2,32,32
54Ø DATA 32,1Ø2,1Ø2,1Ø2,16Ø,1Ø2,
    1Ø2,1Ø2,1Ø2,32,32,1Ø2,32,32
55Ø DATA 99,99,99,99,99,99,99,99,
    99,99,99,99,99,99
```



1. A house on the Acorn from just four DATA lines printer



2. A Spectrum bridge using PLOT and DRAW, plus DATA

Seven lines of DATA are used: these correspond to the seven screen *rows* (lines) which form part of the FOR ... NEXT loop started in Line 3Ø. The fourteen entries within each DATA line correspond to the fourteen *columns* specified in the FOR ... NEXT loop beginning in Line 4Ø.

When the value of X—a screen display code (see your manual)—has been read as instructed in Line 5Ø, it is POKEd into screen memory as directed by Line 6Ø. A corresponding POKE has to be placed into colour memory for the character to show: this is achieved by Line 65. In both these lines, the number 291 starts the graphic at a point towards the centre of the screen, but the actual position is dictated by the values of the variables A and B. Now key in the following lines:

```
22Ø FOR K=1 TO 2ØØØ
23Ø NEXT K
24Ø PRINT "◘▣□ PRESS ANY KEY
   FOR ANOTHER GO"
25Ø GET K$:IF K$="" THEN GOTO 25Ø
27Ø GOTO 1Ø
```

RUN the program and respond to the prompt after the delay. Immediately you will see an 'out of data' error message: there is no more DATA for the reRUN Line 5Ø to access. However, by simply adding:

```
26Ø RESTORE
```

which sends the computer back to the start of the DATA statement, the program can be RUN any number of times.

This program will build up, in stages, a picture of a castle. Type in these lines and RUN them:

```
1Ø PCLEAR4
2Ø PMODE4,1
3Ø PCLS5
4Ø SCREEN1,1
5Ø READ SX,SY
6Ø LINE-(SX,SY),PSET
7Ø FOR K=1 TO 18
8Ø READ X,Y
9Ø LINE-(X,Y),PRESET
1ØØ NEXT K
27Ø GOTO 27Ø
5ØØ DATA 64,16Ø
51Ø DATA 64,6Ø,32,6Ø,48,4Ø,64,6Ø,32,
   6Ø,32,16Ø,11Ø,16Ø,11Ø,12Ø
52Ø DATA 152,12Ø,152,16Ø,228,16Ø,
   228,6Ø,212,4Ø,196,6Ø,228,6Ø
53Ø DATA 196,6Ø,196,16Ø,196,74
```

Lines 1Ø to 4Ø set up the high resolution graphics screen ready for drawing. Line 27Ø keeps it switched on.

Lines 5Ø and 6Ø are unusual. The coordinates of the start point of the castle are read, and a buff line is drawn on the buff background to the start point. Sometimes it's useful to be able to draw these 'invisible' lines, because they can join together visible lines in a continuous programming pattern.

The lines from 7Ø to 1ØØ draw the outline of the castle. By the end of this piece of program 38 items of DATA have been read.

Now add these lines and you'll see some castellations when you RUN the program:

```
11Ø FOR K=1 TO 33
12Ø LET X=X-4
13Ø LINE-(X,Y),PRESET
14Ø IF Y=74 THEN LET Y=78 ELSE
   LET Y=74
15Ø LINE-(X,Y),PRESET
16Ø NEXT K
```

You're probably wondering what has happened to the READ line. This is one occasion when using READ and DATA will not save you work—you'd need 66 items of DATA to define the corners of the castellations, so this way—checking positions—saves you typing all that extra DATA.

These lines will draw some windows:

```
17Ø FOR K=1 TO 8
18Ø READ X,Y
19Ø LINE(X,Y)-(X+4,Y),PRESET
2ØØ LINE(X+2,Y-2)-(X+2,Y+6),
   PRESET
21Ø NEXT K
54Ø DATA 46,8Ø,46,12Ø,21Ø,8Ø,21Ø,
   12Ø,86,9Ø,17Ø,9Ø,8Ø,132,178,132
```

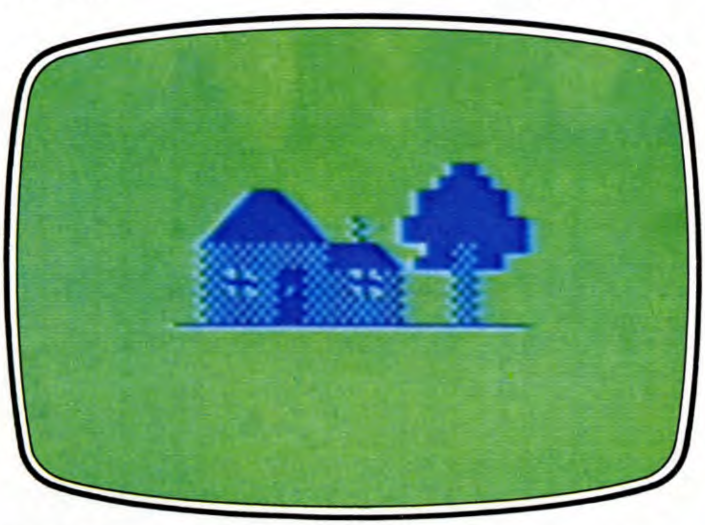
Notice that you don't have to have DATA for each end of the windows, four sets in all. As they are all the same size only one set of DATA is needed per window.

Now try adding these lines. RUN the program and see what happens.

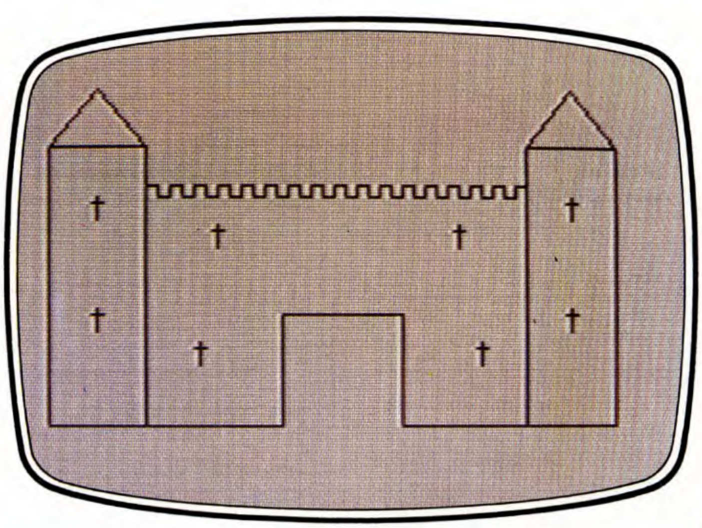
```
22Ø FOR K=1 TO 4ØØØ
23Ø NEXT K
24Ø CLS:PRINT@33,"PRESS ANY KEY
   FOR ANOTHER GO"
25Ø LET IN$=INKEY$:IF IN$=""
   THEN GOTO 25Ø
27Ø GOTO 3Ø
```

When you press a key you will get OD or 'out of data' error. The reason for this is that the program has finished the list of DATA—there is no more. Fortunately you don't have to type in all the DATA again. Just add this line which makes use of the RESTORE statement to instruct the computer to go back to the start of the DATA list.

```
26Ø RESTORE
```



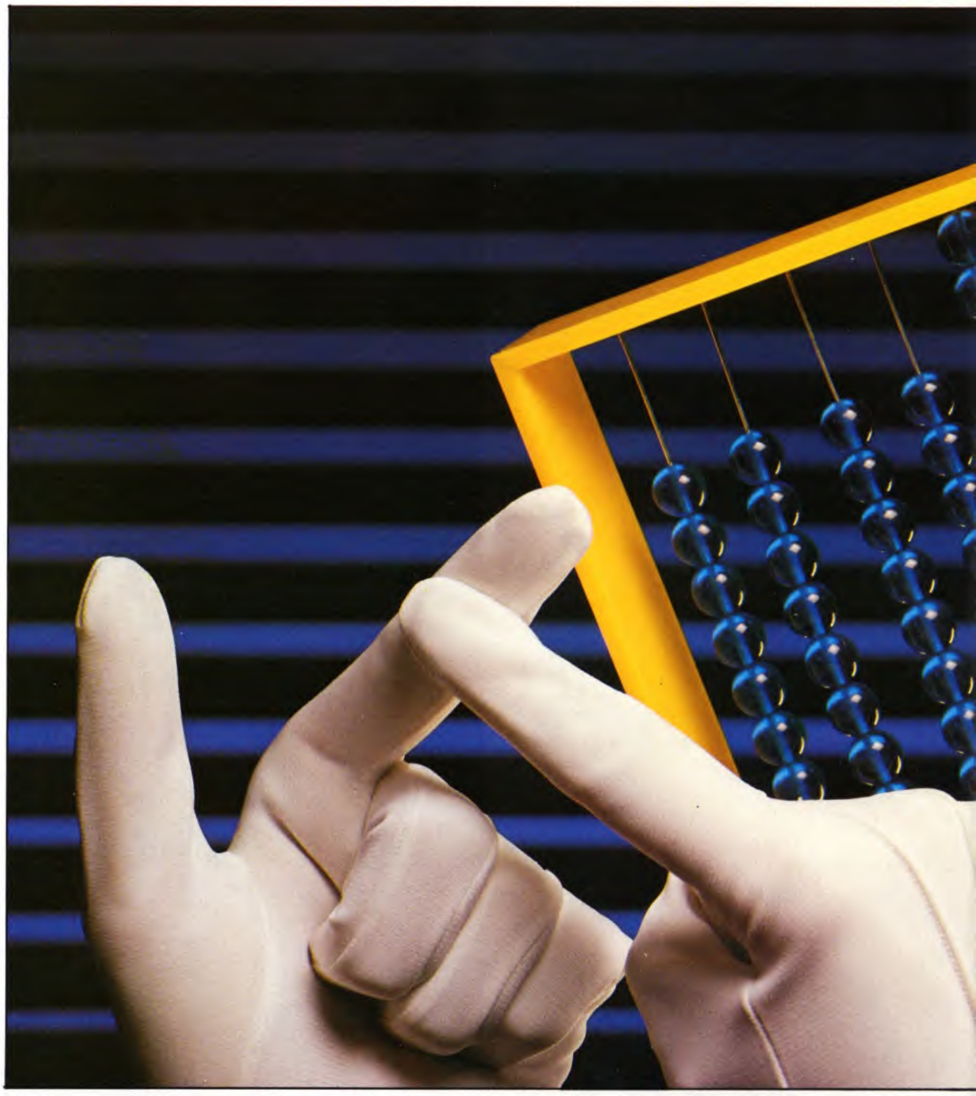
3. ROM graphics on the Commodore called up to order



4. The Dragon/Tandy castle is built up in stages

LEARNING TO COUNT ON ONE FINGER

Computers count in twos—it's as if they had millions of hands but only one finger on each. And for effective machine code programming, it's a system you need to learn as well



One of the problems of learning machine code is that you need to understand a little about the theory of numbers. This is not quite as daunting a task as it may seem. If you can count up to 16 you should have no difficulty. But first of all you have to learn how to count up to two.

WHY BASE TEN?

Even the most unmathematical person has no problem telling the time, working out their change or keeping track of football scores. The use of numbers is such a part of everyday life that we never give a thought to how they work. But when you start programming in machine code you need to take a closer look at the way numbers work.

In the western world we use a number system based on the number 1Ø. That means we start counting in the digits Ø to 9. When we need a number higher than 9, we have used all the available digits. So we put a 1 in the next place to the left and a Ø in the right-hand place. Ten is not represented by a single digit, but by two digits we've used before.

If we go on counting, we go through the digits Ø to 9 again in the right-hand place. And when we've reached 9 and add one more, 1 is again clocked up in the next place to the left. And when that place has filled up to 9 and one more has to be added, a 1 appears in the next place to the left and the two right-hand places reset to Ø.

This is called a *base-ten*, or *decimal*, number system, because the value of the digit is increased by a factor of ten in each place as you count from the right. For example, the number 3,275 is worth $5 + 7 \times 1Ø + 2 \times 1Ø \times 1Ø + 3 \times 1Ø \times 1Ø \times 1Ø$—or $5 + 7Ø + 2ØØ + 3{,}ØØØ$. Each digit increases its worth by ten times its face value for every place it is moved to the left.

All this seems very commonplace because we do it every day without thinking about it. So much so that, in base ten, it seems self evident. But we don't always use base ten.

ANCIENT AND MODERN

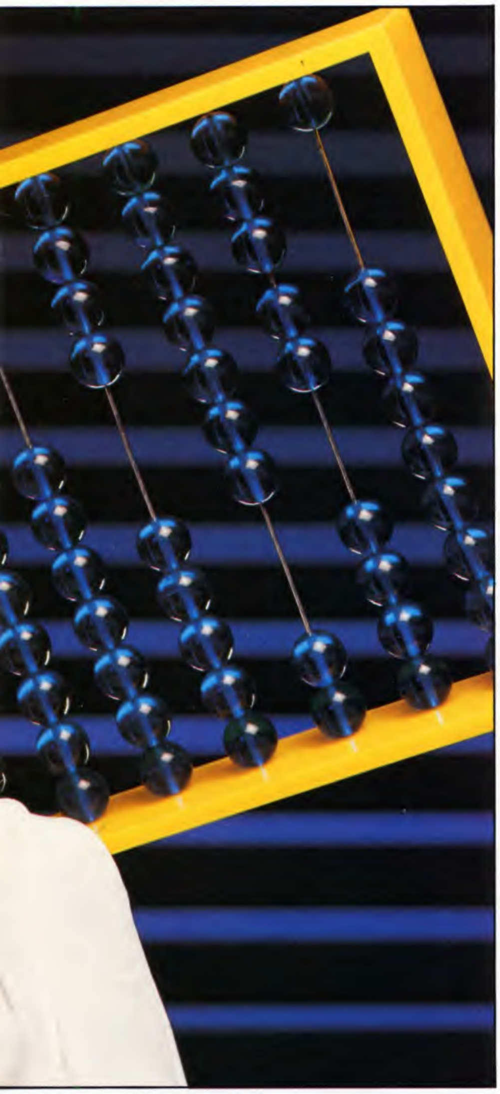
The ancient Babylonians had a system of numbers based on a unit of 6Ø. The vestiges of this can still be seen in our measures of time and angles. There are 6Ø seconds in a minute, 6Ø minutes in an hour, 6Ø minutes of arc in a degree and six times 6Ø degrees—in other words 36Ø—in a complete circle.

You can count up to 59 seconds, but if you add one more you clock up 1 minute and start counting the seconds from Ø again. And when you have 59 minutes and 59 seconds, one more second will give you an hour, while the minutes and seconds places go back to Ø before starting to count up again.

The remnants of number systems with other bases can be seen, too, in the old imperial system of measurement. There are eight pints in a gallon, eight furlongs in a mile, eight stones in a hundredweight and 16 ounces in a pound. Twelve is another base that occurs often. There are 12 inches in a foot, 12 old pennies in a shilling, 12 whatevers in a dozen and 12 dozens in a gross.

Base 12 used to be used in British coinage, and for goods in dozens and grosses, because it was easy to divide by 2, 3, 4 and 6. Money and goods often have to be shared among more than one person and the base 12 made these transactions much easier to perform.

- COUNTING IN BASE TEN
- OTHER NUMBER SYSTEMS
- HOW TO COUNT IN BASE NINE
- DOING SUMS IN DIFFERENT BASES
- A CALCULATOR PROGRAM FOR ALL NUMBER BASES
- THE BINARY SYSTEM
- BITS AND BYTES
- HOW THE COMPUTER ADDS UP



1. Our ten fingers lead us to count in tens, and counting aids like the simple abacus or bead counter are a mechanical model for our fingers.

Ten, on the other hand, is divisible only by two numbers—2 and 5.

It is often convenient to use a number system with a base other than ten. For example, a pound in weight can be broken down into ounces by repeated halving.

NINE OF THE BEST

There is no reason, in fact, why you can't use a system based on any number you like. For example, if we had all been born with one finger fewer on one hand, we should probably have a *nonary*—or nine-based—system. We would be able to count up through the digits from Ø to 8 in the normal way. But if we added one more to 8 it would clock up the digit 1 in the next place to the left and return the first place to Ø. In other words the decimal number nine would be represented in nonary by 1Ø.

Again if 1 was added to 18 you'd get 2Ø, and if 1 was added to 88 you'd get 1ØØ.

As you can see from the example below, the ordinary rules of arithmetic apply whatever number you base your number system on. You can try simple sums in an eight-based or seven-based system, for example. Whatever base you choose to do calculations in, the ordinary rules of arithmetic still apply.

In our nonary system, for instance, as you move each place to the left the face value of the digit is increased by a factor of nine. So 3,275 in nonary is equal to $5 + 7 \times 9 + 2 \times 9 \times 9 + 3 \times 9 \times 9 \times 9$ or 2,417 in decimal. But, you'll note, there is no digit 9 in the nonary system. Nine is represented by 1Ø. So it would be just as correct to say that 3,275 is equal to 5 + 7 × 1Ø + 2 × 1Ø × 1Ø + 3 × 1Ø × 1Ø × 1Ø in nonary.

With a little bit of mental agility you should be able to do simple sums in nonary. For example, 99 × 9 equals 891 in decimal. In nonary, 99 is represented by 12Ø—that is $1 \times 9 \times 9 + 2 \times 9$ or 81 + 18—and 9 is 1Ø.

So in nonary 99 × 9 translates into 12Ø × 1Ø which is 1,2ØØ or $1 \times 9 \times 9 \times 9 + 2 \times 9 \times 9$ or 891 in decimal.

Any addition, subtraction, multiplication or division of nonary numbers will work. Check for yourself. But you have to be a bit careful. Remember, for example that 16 in decimal is 17 in nonary, if you carry or borrow one in nonary it is only worth nine, and in division, 3 goes into 1Ø three times.

PROBLEMS OF THE OVER-TENS

Dealing with number systems with bases greater than ten is a little more difficult though. To handle them easily you have to devise new digits. For example, if you were to use a twelve-based number system you would need to invent digits to represent ten and eleven. In duodecimal, 1Ø would mean 1 × 12, or 12. And 11 would mean 1 × 12 + 1—that is, 13 in decimal.

The easiest way to extend the range of digits is to use the alphabet. Ten could be A, say, and eleven could be B. Some of your duodecimal numbers would look like this: A2, BA and 7B—or 122, 142 and 95 in decimal.

Microtip

A model for numbers in different bases

It can be hard to visualize just how a different number system works, because everyone is so used to counting in tens. After all, it isn't easy to chop off some fingers, or grow extra ones, so you can count on them! But you can, for example, tape two fingers together to try base nine, or carry two extra tally sticks to work in base twelve, and so on.

But a better model is to make your own bead counter—similar to the ones shown on these pages. Just as an ordinary bead counter or abacus can help to clarify and speed up conventional arithmetic in base 1Ø, so you can make a bead frame with as many counters on each line as you want. You don't even need to make a proper frame—just an arrangement of grooves in which you can roll the beads back and forth would be adequate.

Don't forget that you need one less bead in each row than your number base. So in base ten you have nine beads and binary only needs one. No beads slid across equals zero, one bead slid across equals one and so on. When you get beyond nine, you 'carry' a one to the next row, and slide back all the beads to zero.

AN ALL-BASE CALCULATOR

Doing arithmetic in bases other than ten can lead to severe brain strain. So here is a program that will do arithmetic in any base up to 36. When you RUN it, it will ask you which base you want to use. Key the appropriate number in, in decimal.

Next it asks you for a number. This should be entered in the base you have selected. If you've chosen base eight, you shouldn't use any digit over 7, for example. And if you've picked a base greater than ten, digits over 9 should be keyed in using the letters of the alphabet, in capitals. Ten will be A, eleven B, twelve C and so on.

Then the computer will ask you for an arithmetic sign. Key in +,-,* or /. After that you have to supply another number in the same base as before.

If your sum is a division and the answer is not an integer, the computer will tell you that it will not work. 'Won't go' will be PRINTed on the screen. Otherwise the sum and its answer, in the base you've selected, will be displayed on the screen.

```
2Ø INPUT "♥BASE (UP TO 36)";B%
3Ø INPUT "NUMBER (INTEGER)";A$
4Ø INPUT "SIGN";S$
5Ø INPUT "NUMBER (INTEGER)";B$
6Ø PRINT "♥"
7Ø P$=A$:GOSUB 2ØØ
8Ø DA=D%
9Ø P$=B$:GOSUB 2ØØ:DB=D%
1ØØ IF S$="*" THEN X%=DA*DB
1Ø2 IF S$="+" THEN X%=DA+DB
1Ø4 IF S$="−" THEN X%=DA−DB
1Ø6 IF S$="/" THEN X%=DA/DB
12Ø AN$=""
13Ø IF B%*(X%/B%−INT(X%/B%))>9
    THENAN$=CHR$(55+B%*(X%/B%−
    INT(X%/B%)))+AN$:GOTO15Ø
14Ø AN$= RIGHT$(STR$(B%*(X%/B%−
    INT(X%/B%))),1)+AN$
15Ø X%= INT(X%/B%)
16Ø IF X%>Ø THEN GOTO13Ø
18Ø PRINT "♥▦▦▦▦BASE";B%
    ":□"A$S$B$"="AN$
19Ø END
2ØØ D%=Ø:IX%=Ø
2Ø5 Y=ASC( RIGHT$(P$,1))
21Ø IF Y<58 THEN D=Y−48:
    GOTO 23Ø
22Ø D=Y−55
23Ø D%=D%+D*B%↑IX%:
    P$= LEFT$(P$, LEN(P$)−1):
    IX%=IX%+1
24Ø IF LEN(P$)>Ø THEN GOTO 2Ø5
25Ø RETURN
```

On the ZX81, type in the following program entirely in capitals. Omit Line 1Ø completely, omit LINE in Lines 3Ø, 4Ø, and 5Ø, and use ** instead of ↑ in Line 22Ø.

```
1Ø POKE 23658,8
2Ø INPUT "BASE (UP TO 36)□";b
3Ø INPUT "NUMBER (INTEGER)□"; LINE a$
4Ø INPUT "SIGN□"; LINE s$
5Ø INPUT "NUMBER (INTEGER)□"; LINE b$
6Ø CLS
7Ø LET p$=a$: GOSUB 18Ø
8Ø LET x$=STR$ dec
9Ø LET p$=b$: GOSUB 18Ø
95 LET y$=STR$ dec
1ØØ LET z$=x$+s$+y$: LET x=VAL z$
11Ø IF x<>INT x THEN LET n$="WILL
    NOT GO": GOTO 16Ø
12Ø LET n$=""
13Ø LET u=INT (x/b): LET u=u*b: IF
    x−u>9 THEN LET n$=CHR$
    (55+(x−u))+n$: GOTO 14Ø
135 LET n$=STR$ (x−u)+n$
14Ø LET x=INT (x/b)
15Ø IF x>Ø THEN GOTO 13Ø
16Ø PRINT AT 1Ø,Ø;a$+s$+b$;"=□";n$
17Ø STOP
18Ø LET dec=Ø
19Ø LET index=Ø
2ØØ LET y=CODE p$(LEN p$)
21Ø IF y<58 THEN LET d=y−48:
    GOTO 22Ø
215 LET d=y−55
22Ø LET dec=dec+d*b↑index
23Ø LET p$=p$(1 TO (LEN p$)−1)
24Ø LET index=index+1
25Ø IF LEN p$>Ø THEN GOTO 2ØØ
26Ø RETURN
```

```
1Ø MODE6:PRINT"
2Ø INPUT "BASE (UP TO 36)",B%
3Ø INPUT "NUMBER (INTEGER)",A$
4Ø INPUT "SIGN",S$
5Ø INPUT "NUMBER (INTEGER)", B$
6Ø CLS:PRINT"
7Ø P$=A$:GOSUB18Ø
8Ø DECA$=STR$(DEC%)
9Ø P$=B$:GOSUB18Ø: DECB$=
    STR$(DEC%)
1ØØ X%=EVAL(DECA$+S$+DECB$)
11Ø IF X%<>EVAL(DECA$+S$+DECB$)
    THEN AN$= "WON'T GO":
    GOTO 16Ø
12Ø AN$=""
13Ø IFX%MODB%>9THEN AN$=CHR$
    (55+X%MODB%)+AN$ ELSEAN$=
    STR$(X%MODB%)+AN$
14Ø X%=X%DIVB%
15Ø IFX%>Ø GOTO 13Ø
16Ø PRINTTAB(Ø,12) A$+S$+B$+
    "="+AN$
17Ø END
18Ø DEC%=Ø
19Ø INDEX%=Ø
2ØØ LET Y=ASC(RIGHT$(P$,1))
21Ø IF Y<58 THEN D=Y−48
    ELSE D=Y−55
22Ø DEC%=DEC%+D*B%∧INDEX%
23Ø P$=LEFT$(P$,LEN(P$)−1)
24Ø INDEX%=INDEX%+1
25Ø IF LEN(P$)>Ø THEN GOTO 2ØØ
26Ø RETURN
```

```
1Ø CLS
2Ø INPUT"BASE (UP TO 36) ";B
3Ø INPUT"NUMBER (INTEGER)□";A$
```

2. Any number system can be modelled mechanically—the clock is in base 60, the bead frame in base 10



```
40 INPUT"SIGN□";S$
50 INPUT"NUMBER (INTEGER)□";B$
60 P$=A$:GOSUB 210
70 X$=STR$(DE)
80 P$=B$:GOSUB 210
90 Y$=STR$(DE)
100 IFS$="*" THENX=VAL(X$)
    *VAL(Y$)
110 IFS$="/" THENX=VAL(X$)
    /VAL(Y$)
120 IFS$="+" THEN X=VAL(X$)
    +VAL(Y$)
130 IFS$="-" THENX=VAL(X$)
    -VAL(Y$)
140 IFX<>INT(X) THENN$="WON'T
    GO":GOTO190
150 U=B*INT(X/B):IF X-U>9
    THEN N$=CHR$(55+(X-U))
    +N$:GOTO170
160 N$=MID$(STR$(X-U)+N$,2)
170 X=INT(X/B)
180 IFX>0 GOTO150
190 PRINT@257,A$+"□"+S$+
    "□"+B$;"=□";N$
200 END
210 DE=0
220 IN=0
230 Y=ASC(RIGHT$(P$,1))
240 IFY<58 THEN D=Y-48:GOTO260
250 D=Y-55
260 DE=DE+D*B↑IN
270 P$=LEFT$(P$,LEN(P$)-1)
280 IN=IN+1
290 IFLEN(P$)>0 THEN GOTO230
300 RETURN
```

If you find the whole idea of doing arithmetic in different bases more than you feel like handling, note that even the computer cheats. This program actually converts the numbers you've INPUT into decimal, does the arithmetic, then converts the answer back into the base you've selected. Unlike the flexible human mind, your computer can do arithmetic only in decimal in a BASIC program. (This, despite converting the numbers into binary when it translates the instructions into machine code to perform them!)



AND NOW, BINARY

In computers the most convenient number system to use is that based on two. This is because a computer is composed of electronic switching circuits which have two obvious states—off and on. The off state represents the digit Ø and the on state the digit 1. And in base two that is all the digits you need.

The base-two number system is known as *binary*. It is composed of nothing but Øs and 1s, all the other digits have been abolished. So if you start counting from Ø you can get as far as 1 in the normal way. Add 1 to 1, and 1 is clocked up in the next place to the left while the first place returns to Ø. So, in binary, 1+1=1Ø.

Counting through from zero to eight gives Ø, 1, 1Ø, 11, 1ØØ, 1Ø1, 11Ø, 111, 1ØØØ. Again these numbers obey all the ordinary laws of arithmetic. For example: if you add 1Ø+11, you first add the two right-hand digits, Ø+1. Then you add the two left-hand digits, 1+1 which gives two, but two in binary is 1Ø. So 1Ø+11=1Ø1—in other words, 2+3=5.

Subtraction in binary is a similarly straightforward process. The only thing to watch is when to carry.

The beauty of binary is that even for human beings multiplication and division are extraordinarily easy. At each stage of a multiplication you are either multiplying Ø by Ø (which gives Ø), Ø by 1 (which again gives Ø) or 1 by 1 (which gives 1).

In binary, in fact, multiplication is so easy that even a computer can do it.

Long division is similarly simple. At each stage you only have to decide whether the divisor goes into the number that is being divided once or no times.

Check for yourself that all the normal arithmetic processes work in binary. The panel on the next page gives a series of worked examples of addition, subtraction, multiplication and division for you to try.

BITS AND BYTES

Binary numbers accurately reflect what is going on in the computer. Every instruction or piece of DATA you feed into the computer is encoded in binary which the switching circuits of the computer manipulate and store. When you understand binary you begin to understand how the computer works.

Each of the digits of a binary number is represented electronically in the computer by a circuit that is on or off. If it is on, the value is 1. If it is off, the value is Ø. Each digit circuit is known as a bit.

To memorize and manipulate binary numbers these circuits are organized into larger entities which can represent larger, more useful numbers. In nearly all home computers the bits are organized into groups of eight, giving what is known as an eight-bit byte. That means each byte can represent eight binary digits, so it can store any number between ØØØØØØØØ or zero, and 11111111, or 128+64+32+16+8+4+2+1 which is 255. Numbers larger than 255 are represented by two bytes or more.

BINARY FRACTIONS

All the programs and examples shown so far have all dealt with whole numbers, but it is also possible to convert fractions into binary numbers.

For example, $\frac{1}{2}$ is Ø.1 in binary, $\frac{1}{4}$ is Ø.Ø1, $\frac{1}{8}$ is Ø.ØØ1. You may already be able to see a pattern emerging. In fact, any fraction can be made up of $\frac{1}{2}$s, $\frac{1}{4}$s, $\frac{1}{8}$s, $\frac{1}{16}$s and so on—just as any whole number can be made up of 1s, 2s, 4s, 8s etc. It's just that it is more difficult to work out the binary equivalent of fractions because the series of Øs and 1s is often infinite!

When you're converting fractions, you can think of the $\frac{1}{2}$s, $\frac{1}{4}$s etc. as .5, .25 and so on. Then you can fairly easily work out that a fraction such as .75 is made up of 1*.5+1*.25, which means its binary equivalent is Ø.11.

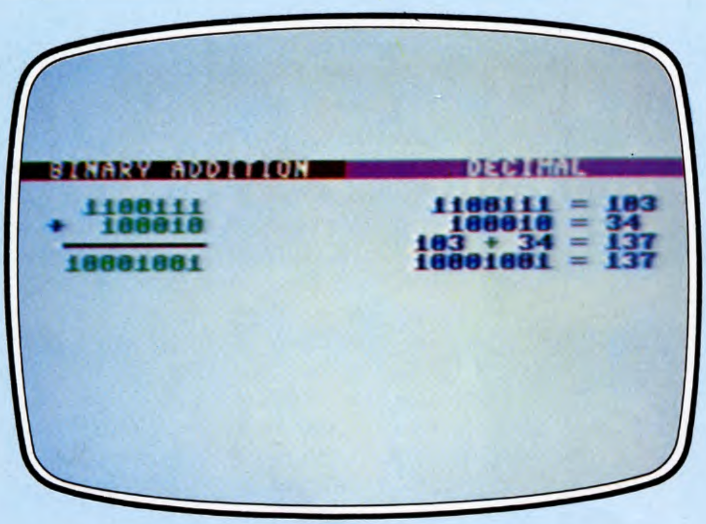
The next program gets the computer to work them out for you:

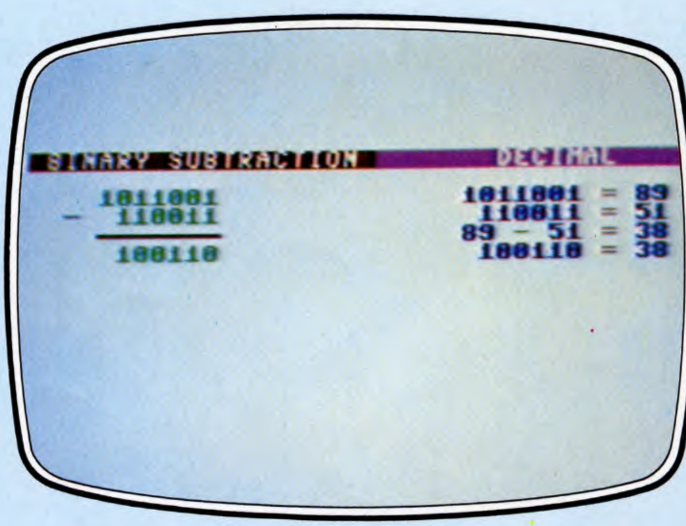
```
1Ø CLS
2Ø PRINT@35,"DECIMAL TO BINARY
   CONVERSION"
3Ø PRINT: INPUT"ENTER A NUMBER
   BETWEEN Ø AND 1 ";N
4Ø IF N<=Ø OR N>=1 THEN 3Ø
5Ø N$="Ø."
6Ø FOR T=1 TO 3Ø
7Ø N=N*2
8Ø N$=N$+CHR$(48+INT(N))
9Ø N=N-INT(N)
1ØØ NEXT T
11Ø PRINT@257,"THE BINARY NUMBER
   IS:-"
12Ø PRINT: PRINTN$: PRINT
13Ø PRINT"ANOTHER NUMBER (Y/N) ?"
14Ø A$=INKEY$
15Ø IF A$="Y" THEN 1Ø
16Ø IF A$<>"N" THEN 14Ø
17Ø END
```

```
2Ø PRINT "♥DEC TO BIN
   CONVERSION"
3Ø PRINT "▣ENTER A NUMBER
   BETWEEN Ø-1";INPUT N
4Ø IF N<=Ø OR N>=1 THEN 2Ø
5Ø N$="Ø."
6Ø FOR T=1 TO 32
7Ø N=N*2
8Ø N$=N$+CHR$(48+INT(N))
9Ø N=N-INT(N)
1ØØ NEXT T
11Ø PRINT "▣THE BINARY NUMBER
   IS:-"
12Ø PRINT "▣"N$
13Ø PRINT "▣ANOTHER NUMBER
   (Y/N) ?"
14Ø GET A$
15Ø IF A$="Y" THEN RUN
16Ø IF A$<>"N" THEN 14Ø
17Ø PRINT "♥" : END
```

```
1Ø CLS
2Ø PRINT '"DECIMAL TO BINARY
   CONVERSION"
3Ø INPUT "ENTER A NUMBER
   BETWEEN Ø AND 1",N
4Ø IF N<=Ø OR N>=1 THEN GOTO 3Ø
45 PRINT '"DECIMAL:- □";N
5Ø LET N$="Ø."
6Ø FOR T=1 TO 16
7Ø LET N=N*2+1E-9
8Ø LET N$=N$+CHR$ (48+INT N)
9Ø LET N=N-INT N
1ØØ NEXT T
11Ø PRINT '"THE BINARY NUMBER
   IS:-"
12Ø PRINT N$
13Ø PRINT ''''"ANOTHER NUMBER
   (Y/N) ?"
14Ø LET A$=INKEY$
15Ø IF A$="Y" THEN GOTO 1Ø
16Ø IF A$<>"N" THEN GOTO 14Ø
17Ø STOP
```

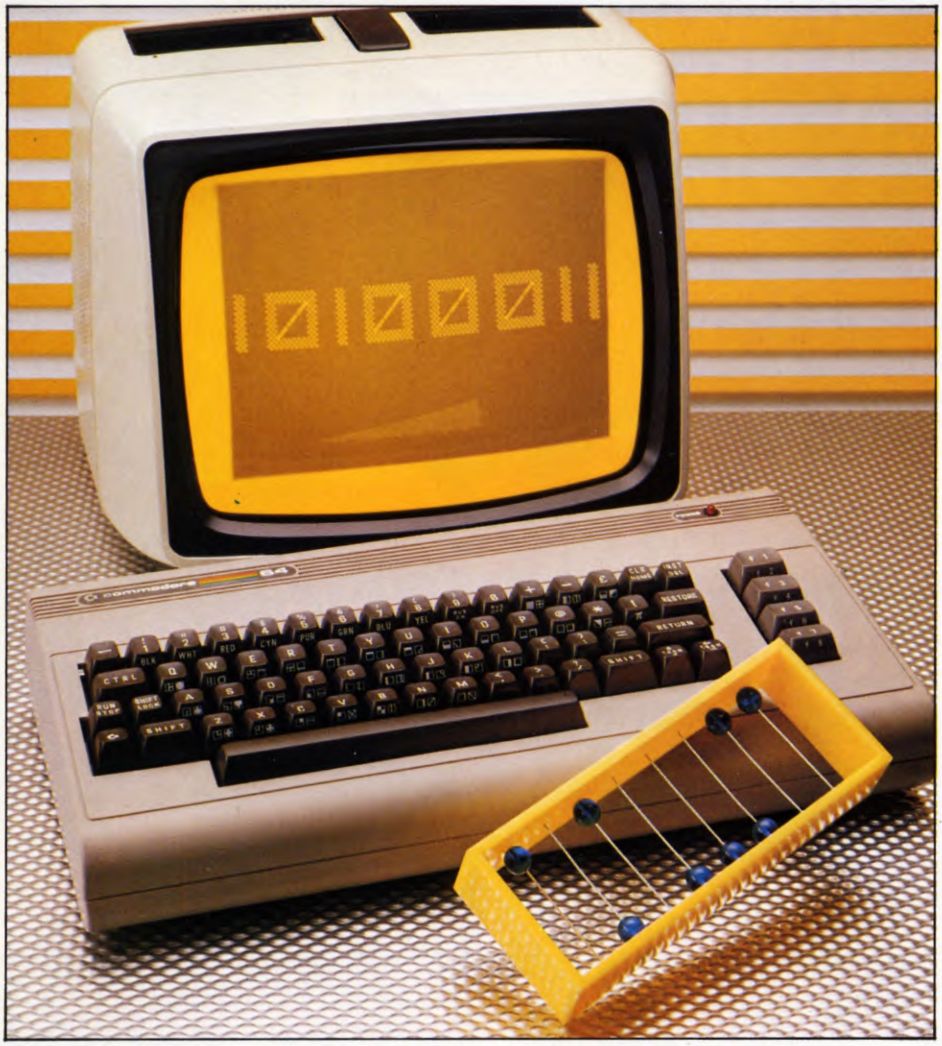
```
1Ø MODE 1
2Ø PRINT'"DECIMAL TO BINARY
   CONVERSION"
3Ø INPUT'"ENTER A NUMBER
   BETWEEN Ø AND 1",N
```





1. Binary addition works from the right. Ø+Ø gives Ø. Ø and 1 give 1. 1 and 1 give 1Ø, which is Ø and carry 1. Check how this works through the sum above

2. Binary subtraction is similarly simple. 1−1 gives Ø. Ø from 1 gives 1, while if you try to take 1 from Ø, you must carry this over to the next digit to the left

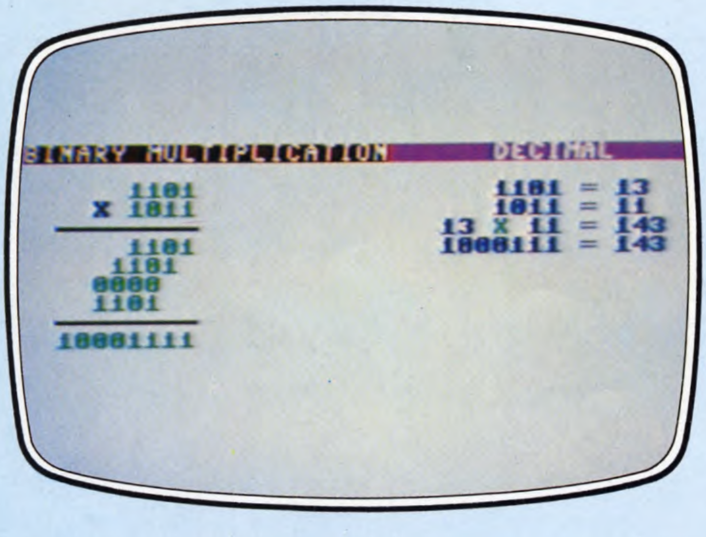


```
40 IF N< =Ø OR N> =1 THEN 3Ø
5Ø N$="Ø."
6Ø FO T=1 TO 32
7Ø N=N*2
8Ø N$=N$+CHR$(48+INT(N))
9Ø N=N-INT(N)
1ØØ NEXT
1ØØ PRINT''"THE BINARY NUMBER IS :-"
12Ø PRINTN$
13Ø PRINT'''"ANOTHER NUMBER (Y/N)?"
14Ø A$=GET$
15Ø IF A$="Y" THEN 1Ø
16Ø IF A$< >"N" THEN 14Ø
17Ø END
```

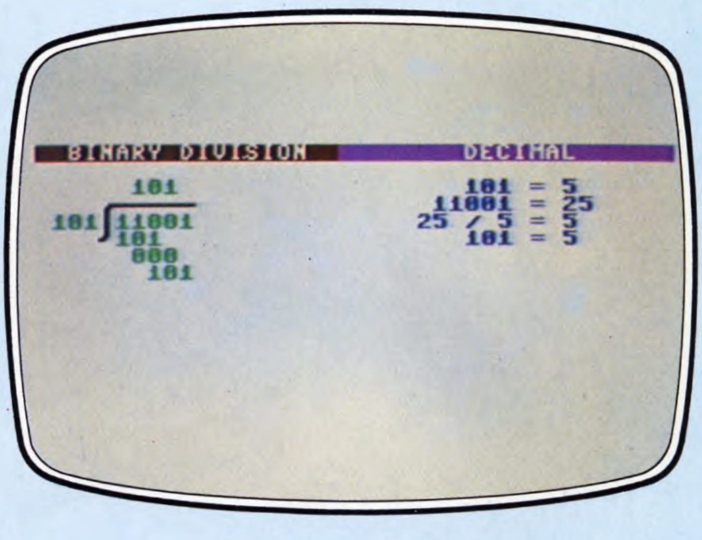
The program accepts any number between Ø and 1 and then PRINTs out its binary equivalent. Line 7Ø starts by doubling your number then Line 8Ø builds up the binary number one character at a time. The computer first works out the value of INT(N), which will be either Ø or 1, then this is added to 48 to make either 48 or 49—which are simply the ASCII codes for "Ø" and "1". All this is doing in effect is to turn the numbers into strings so they can be joined on to N$. Line 9Ø takes the integer away from N to leave just the fractional part.

The Sinclair program is slightly different because while INT 1 is 1 as you would expect, INT (.5*2) comes out as Ø. This is because it stores .5*2 as .99999999.... So a small fraction has to be added on to N in Line 7Ø to tip the balance and make INT work as it should.

3. A binary abacus only needs single counters—in a computer, pulses of electricity in a circuit can do the job

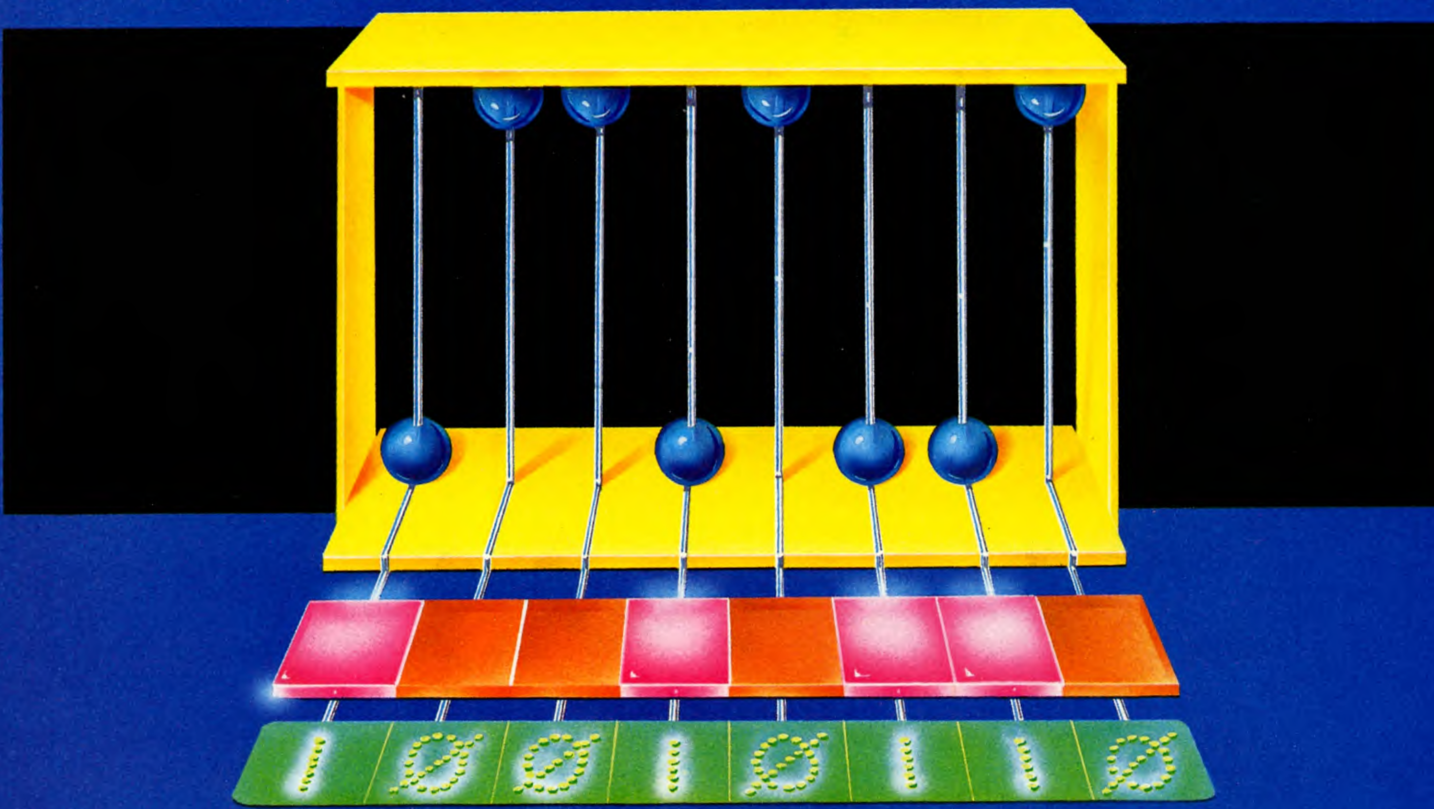


3. At each stage of a binary multiplication, you either multiply Ø by Ø, which gives Ø, Ø by 1, which also gives Ø, or 1 by 1, which gives 1. See how this applies in the example above



4. In binary long division you have to check whether the divisor goes into the number you are dividing once, in which case you enter 1, or no times, when you enter a Ø

FROM THE ABACUS TO ELECTRONICS





1 Ø 1 1 1 Ø 1 1 - Ø 1 Ø Ø 1 1 Ø 1 = Ø 1 1 Ø 1 1 1 Ø



Ø 1 1 1 1 1 1 1 + Ø Ø Ø Ø Ø Ø Ø 1 = 1 Ø Ø Ø Ø Ø Ø Ø



1 1ØØ111 111Ø1 1 11Ø - Ø111Ø11ØØØ11ØØ11 = Ø1Ø1 1ØØ1 1Ø1Ø1Ø1 1

Binary numbers can be added, subtracted, multiplied and divided like any other numbers, but were chosen for use in computers for their unique property—it is easy to tell if a circuit is on, or if it is off.

Binary digits are represented by computer circuits. When a circuit is off it represents a Ø, when it is on it represents a 1. In the chips in your home computer the binary circuits are arranged in eights so eight-bit binary numbers can be represented. This gives a range of ØØØØØØØØ, or Ø in decimal, to 11111111, or 255 in decimal. If you look through your computer's manual you will find the number 255—or 256 if you start counting from 1 instead of Ø—comes up over and over again.

For some purposes larger numbers are required and two eight-bit binary memory locations are taken together to represent a 16-bit binary number. The Dragon microprocessor can handle these and sometimes does its arithmetic in 16-bit binary. In this case, numbers between ØØØØØØØØØØØØØØØØ and 1111111111111111, or Ø and 65,535 in decimal, can be handled.

SPOT-ON SCREEN DISPLAYS

- USING **TAB** WITH **PRINT** AND **INPUT**
- POSITIONING WITH PUNCTUATION MARKS
- MAKING A NEAT DISPLAY
- HOW TO USE SPACING

Good screen display makes your instructions readable and your tables understandable. It's all a matter of using the right commands

PRINT and INPUT are two of the first commands that programmers learn. They are used all the time and in all sorts of programs and you should have a fairly good idea of how they work. But you may not be using them to their full advantage.

There are many ways of laying out text on the screen, and it is important to make a good display, especially if you write programs that you want other people to use. If the screen is too full of confusing instructions or the text is all jumbled up, then many people will become impatient and won't take the trouble to read it. Your carefully worked out quiz, chart or game will then be useless.

So whatever sort of program you are writing, the text and instructions must be easy to read, and if you want someone to INPUT information it must be obvious exactly what you want.

Each computer has its own way of positioning text on the screen. The commands are usually TAB, AT, the @ symbol, or the cursor control characters. Punctuation—commas, semicolons and apostrophes—are also used to separate different PRINT statements.

This article explains how to make full use of all these commands so you can make neat and readable displays that are understandable at a glance.

You probably know that a command like this:

```
PRINT "Good morning"
```

will PRINT the message on the first clear line starting at the far left. If you clear the screen first:

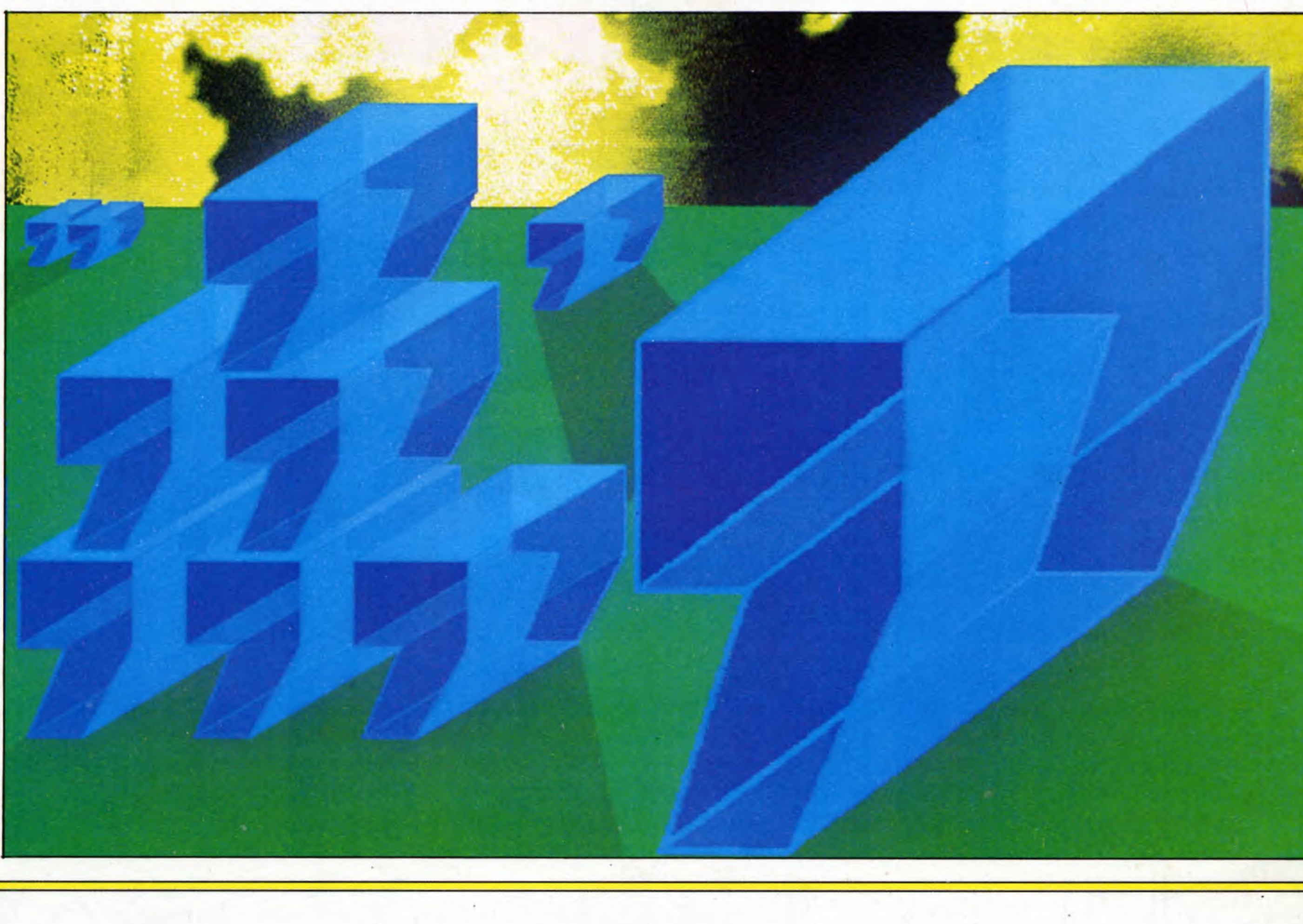
```
10 CLS
20 PRINT "Good morning"
```

it will PRINT it on the top line of the screen. The computer is doing just the same as you would if you were given a clean sheet of paper.

But you can tell the computer to PRINT the message at a different position. Add this line to the last program:

```
30 PRINT TAB(20) "Good afternoon"
```

This PRINTS out the words starting at column 2Ø, halfway across the screen. Each word can be positioned separately if you want, using two TAB statements.





Try changing Line 3Ø to:

```
3Ø PRINT TAB(2Ø) "Good" TAB(3Ø)
   "afternoon"
```

This now PRINTS the first word at column 2Ø and the second word at column 3Ø.

Note that there is *no* space between the keyword TAB and the first bracket. If you did leave a space the computer would not understand what you wanted and would PRINT a 'no such variable' error.

If you want the message PRINTed on a different line then you must put two numbers in the brackets. For example, this next line PRINTs the words at column 2Ø as before, but this time on line 15:

```
4Ø PRINT TAB(2Ø,15) "Good night"
```

TAB with just one number is useful after you've already used one TAB command to move to a different line. For instance, to position two words (or numbers) on a single line, first move to the first item with TAB(column, line) then PRINT the second item at TAB(column)—you are already on the correct line.

But note: in this case the number in the second TAB is the number of columns from the last one, *not* the absolute position on the screen. So this line from a program is quite acceptable:

```
5Ø PRINT TAB(2Ø,17) "first" TAB(8)
   "second"
```

The word "second" is PRINTed 8 columns after the word "first", at column 28, not at column 8 as you might think at first.

If you tell the computer to PRINT beyond the end of the line—TAB(85) for instance—it just counts on to the next line or even several lines until it has moved the correct number of columns.

It is worth experimenting with the TAB command to get a better feel for what it will and will not do.

INPUTTING INFORMATION

TAB can be used with INPUT as well as with PRINT. Here's an example:

```
1Ø INPUT TAB(1,5)"Please enter your
   name□",name$ TAB (1,6)"and
   your age□",age
2Ø PRINT TAB(1,1Ø) "Name□";name$;
   "□age□";age
```

This will PRINT the first instruction at column 1, line 5 and then prompt you with a question mark for an INPUT. Whatever you type is stored as 'name$'. The next instruction is PRINTed on the next line exactly below the first and another question mark prompts you for the second INPUT which is stored as 'age'. This is a numeric variable so you must type in a number. Line 2Ø simply PRINTs out name and age so you can see that the information has actually been stored.

If you don't want the question mark, simply miss out the commas in Line 1Ø. This would be necessary in a line such as:

```
INPUT "Cost of item £" cost
```

as a question mark would spoil the display. 'Cost of item £67.45' is obviously much better than 'Cost of item £?67.45'.

Here's a short program that makes use of INPUT TAB. It allows you to enter cricket scores and automatically tabulates them into neat columns on the screen. There's no need to work out the batsman's average for the last column—the computer does this for you and PRINTs out the result as soon as you have entered the other figures.

```
1Ø CLS
2Ø PRINT' TAB(1) "NAME" TAB(13)
```

```
    "INNINGS" TAB(22) "RUNS" TAB(29)
    "N.O." TAB(35) "AV"
3Ø LET ROW=3
4Ø INPUT TAB(1,ROW) N$
5Ø INPUT TAB(13,ROW) I
6Ø INPUT TAB(22,ROW) R
7Ø INPUT TAB(3Ø,ROW) N
8Ø IF I<>N THEN PRINT TAB(35,ROW);
    INT((R/(I-N))*1ØØ)/1ØØ ELSE
    PRINT TAB(35,ROW)"*"
9Ø LET ROW=ROW+1
1ØØ GOTO 4Ø
```

The average is worked out in Line 8Ø. It looks rather complicated but this is only so the result is given to two decimal places. The straightforward average is runs divided by (innings minus 'not outs') but this would be calculated to 9 decimal places. This is much too accurate for our needs, and would also mess up the display as the numbers would run into each other.

One other complication is that if the batsman has as many 'not outs' as innings you will get an error report—'division by zero'—when it tries to work out the average. So Line 8Ø also checks whether I equals N, and if it does then it PRINTs out an asterisk in place of the average.

USING PUNCTUATION MARKS

As you have seen, TAB is used to position text and get INPUTs at any position on the screen. But there are other ways of laying out or displaying information without using TAB. It is simply a matter of using the correct punctuation—either commas, semicolons or apostrophes. When you get the hang of these tricks you'll find that in a lot of cases, especially with columns of figures or words, the information can be positioned very easily with hardly any effort on your part and only the occasional TAB statement.

Try these next few lines to get a feel of how it works:

```
5 CLS
1Ø PRINT "Ø123456789Ø123456789
    Ø123456789Ø123456789"
2Ø PRINT 9;9;9
3Ø PRINT 9,9,9
4Ø PRINT 9'9'9
```

(Line 1Ø just numbers the columns along the top of the screen for reference.) Notice carefully how the punctuation affects where the numbers are PRINTed. The numbers separated by a semicolon are PRINTed next to each other. A comma separates the numbers into 'fields', each 1Ø columns wide. And an apostrophe puts each number on a new line.

Now try the same program with letters (that is, strings) instead of numbers. Add these lines and RUN the program again:

```
45 LET A$="A"
5Ø PRINT A$;A$;A$
6Ø PRINT A$,A$,A$
7Ø PRINT A$'A$'A$
```

Here you'll notice that the punctuation marks do the same thing but that strings are always PRINTed at the left of each 1Ø-column field while numbers are PRINTed at the right.

This does actually make sense. If you wrote out a column of words you would normally line them up at the left, but you'd almost certainly line up a column of figures on the right so the units, tens, hundreds, and so on were all in line. And this is just what the computer does.

Type in and RUN this next program. See how the computer lines up the figures and then adds them up—just as you would if you were doing the sum on paper.

```
1Ø PRINT 14'67'245'42'811
2Ø PRINT '14+67+245+42+811
```

If you want several columns then use the comma. Here's a program to PRINT out the square, cube and fourth power of all numbers from 1 to 1Ø.

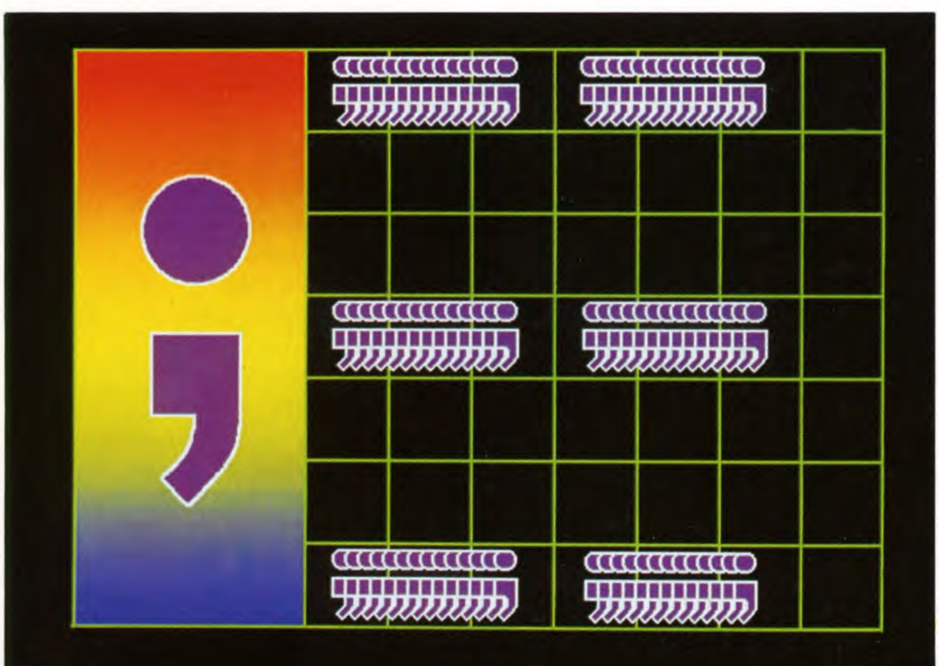
```
1Ø CLS
2Ø PRINT' TAB(4)"NUMBER" TAB(14)
    "SQUARE" TAB(26)"CUBE" TAB(31)
    "4TH POWER"
3Ø FOR J=1 TO 1Ø
4Ø PRINT J,J*J,J*J*J,J*J*J*J
5Ø NEXT J
```

The numbers are PRINTed out at Line 4Ø and the commas make sure that each number is lined up in a separate field. Note that the headings to the columns do need to be positioned using TAB. If you used commas here, the headings would be lined up at the left of each field and would not be above the numbers.

The semicolon is also extremely useful. It is often used after a TAB statement so that what comes next is PRINTed without any spaces in between. The semicolon is not actually necessary if you are PRINTing out strings because, as you know by now, strings are always PRINTed to the left anyway. But it is essential when you're PRINTing numbers and you must put it in if you want the numbers PRINTed at the TAB position.

The semicolon is also useful at the end of a PRINT statement as it keeps the cursor there ready for the next PRINT item—which may come some time later. This last program PRINTs out the alphabet using a FOR ... NEXT loop to step through the ASCII codes for the characters from A to Z:

```
1Ø CLS
2Ø FOR code=65 TO 9Ø
```



1. The semicolon can be used with PRINT or INPUT to control how the information is displayed. Each item is put on the screen immediately following the one before, on the same line

```
30 PRINT CHR$(code);
40 NEXT code
```

Try it without the semicolon and you'll find that each letter is PRINTed on a separate line.

The punctuation marks and TAB offer such a variety of ways to display text on the screen that you're sure to find some combination that suits your particular program. It just takes a bit of practice to know exactly what each one can do.

The Sinclair machines have three statements to make things on the screen look neater: PRINT, PRINT AT and PRINT TAB.

PRINT, preceded by a line number and followed by nothing at all, is the simplest. It just PRINTs a blank line. A couple of examples are in the cricket program (see below).

PRINT AT you have probably met before but it deserves a slightly fuller explanation:

The Spectrum screen has 22 lines, numbered Ø–21 from the top of the screen. It has 32 'columns', or character places, numbered Ø–31 from the left of the screen. The line number is always given first. So these lines:

```
PRINT AT Ø,Ø;"*"
PRINT AT 21, Ø; "*"
PRINT AT Ø, 31; "*"
PRINT AT 21, 31; "*"
```

will PRINT a star at each corner of the screen.

If you tell the computer to PRINT more than one character at a particular screen location, it will PRINT the first character where you tell it to, then just carry on in the same line.

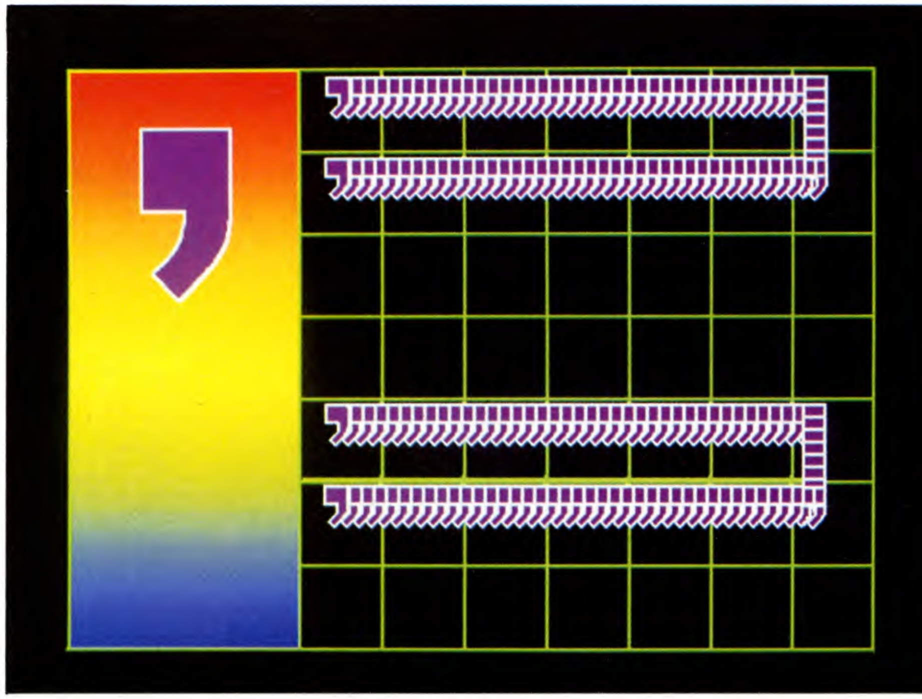
```
PRINT AT 1Ø, 12; "LENGTH"
```

. . . for example, puts the 'L' at 1Ø, 12 and the other letters at 1Ø, 13 to 1Ø, 17 respectively.



This screen picture (this example was generated on the Spectrum) shows you how you can use a semicolon to join up your screen messages. Try a program like:

```
10 PRINT "message";
20 GOTO 1Ø;
```



2. Apostrophes are not used on all machines. When they are available, they instruct the computer to put the next item on a new line, when it would not do so automatically

And if what you want displayed will not fit in the line, starting from the location you specify, it will simply overspill into the line below—sometimes changing the meaning in the process. Try this, for instance:

```
PRINT AT 5, 29; "TENANTS"
```

PRINT TAB

The PRINT TAB instruction on the Spectrum works in much the same way as PRINT AT. There are two differences, though:

1. You do not have to specify the line.

2. You cannot use PRINT TAB to overwrite—and therefore erase—any material on the screen. If you enter a new PRINT TAB at the old position the display drops down a line.

To make the difference clear in your mind, first try this:

```
PRINT AT Ø, 15; "*"
PRINT AT Ø, 15; "?"
```

Then clear the screen (CLS) and try this:

```
PRINT TAB 15; "*";
PRINT TAB 15; "?";
```

For reasons explained below, *be very careful with the punctuation.*

PRINT TAB has no great advantage over PRINT AT—until, that is, you have a lot of tabulated material to enter. Then it saves a lot of typing, and also having to remember line numbers as you go.

This little program, for example, allows you to enter a series of cricket scores and tabulates them neatly on the screen. It also works out the batsman's average. Enter all the lines in capitals on the ZX81:

```
10 PRINT TAB 14;"INNS";TAB 19;"RUNS";
   TAB 24;"N.O.";TAB 29;"AVG"
15 PRINT
20 INPUT "Name?";n$
30 PRINT TAB Ø;n$;
40 INPUT "Innings?";i
50 INPUT "Runs?";r
60 INPUT "Not outs?";n
80 PRINT TAB 14;i;TAB 19;r;TAB 24;
   n;TAB 29;INT (r/(i—n))
90 PRINT
100 PAUSE 200
110 GOTO 20
```

The program looks neat on the screen partly because of the tabulation and partly because Line 9Ø PRINTs a blank line under each batsman's figures.

It is also easy to use. When you first RUN it, it will first ask you for each batsman's name, then his other details. When it asks for 'Runs?' there is no need to perform feats of mental agility to add his newest score to his previous total—you just let the computer do the work.

There are two snags in the program as it stands, though. The first is that the average calculated by the last bit of Line 8Ø is always

rounded *down* to a whole number. How to deal with such number complications is in a later article, but if this average is too coarse for you you can always enter a separate PRINT TAB line for the average and have it calculated to ten decimal places!

The second 'catch' is that should a batsman have as many 'not outs' as innings you will get an error report: Number too big. This is because this tail-ender's theoretical average is actually infinity!

To deal with the problem, delete the existing Line 8Ø and add these new lines:

```
7Ø IF i=n THEN PRINT TAB 14;i;TAB
   19;r;TAB 24;n;TAB 29;"*"
8Ø IF i<>n THEN PRINT TAB 14;i;TAB
   19;r;TAB 24;n;TAB 29;INT (r/(i-n))
```

To escape from the program without clearing the screen, you will need to press CAPS SHIFT and BREAK together (BREAK alone on the ZX81). Line 1ØØ gives you time to do this before the computer demands another name.

A program to tabulate and calculate at the same time is also quite straightforward. Here is one simple example, the raw beginnings of a household accounts program:

```
1Ø PRINT TAB Ø;"DATE";TAB 1Ø;"ITEM";
   TAB 26;"COST"
15 PRINT : PRINT
2Ø LET total=Ø
3Ø FOR I=1 TO 8
4Ø INPUT "Date?";d$
5Ø INPUT "Item?";i$
6Ø INPUT "Cost?";c
7Ø PRINT TAB Ø;d$;TAB 1Ø;i$;TAB 26;c
75 PRINT
8Ø LET total=total+c
9Ø NEXT I
11Ø PRINT TAB 26;"- - - - - - -";
12Ø PRINT TAB 1Ø;"Total";TAB 26;total
```

WATCHING THE DOTS

All your efforts to make a neat and tidy screen display will instantly be spoiled if you get the punctuation wrong, especially with a PRINT TAB statement.

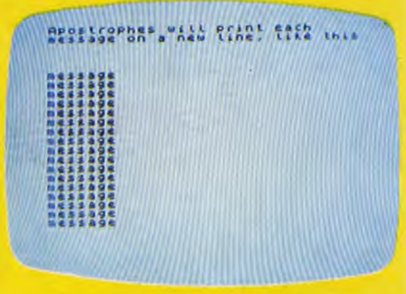
Normally, two sets of semicolons are used after a PRINT TAB statement. To see why, try this tiny program:

```
1Ø FOR n= 1 TO 21
2Ø FOR t=Ø TO 24 STEP 6
3Ø PRINT TAB t; n;
4Ø NEXT t
5Ø NEXT n
```

When you have RUN it, change Line 3Ø to:

```
3Ø PRINT TAB t, n;
```

. . . and then compare it to this:



Another actual screen example shows the force of the apostrophe in a program line like:

```
1Ø PRINT "message" '
2Ø GOTO 1Ø
```

```
3Ø PRINT TAB t; n
```

You can puzzle out, if you wish, why you get the results you do, but it is not important. What is important are these rules:

1 A semicolon means, 'close the next word (or whatever) up tight, without a space'.

2 A comma means, 'start the next word (or whatever) at the beginning of column Ø or column 15, whichever comes first'.

3 And a PRINT TAB statement needs *two* sets of semicolons, unless you want a terrible mess!

The easiest way to display a message on the screen is like this:

```
PRINT "HELLO"
```

'Hello' will appear at the far left of the screen on the next available line—if PRINT "HELLO" is on the last line of the screen the screen will scroll to make room for the message. You must be careful to type the inverted commas or else you will get an error report.

In this form all the programming clutter remains on the screen as well as the message. It would be better if the message appeared on a clear screen. Here is what you need to do:

```
1Ø CLS
2Ø PRINT "HELLO"
```

It's just like giving the computer a clean piece of paper to write on. 'Hello' now appears in the top left on an otherwise blank screen. Add this line and RUN the program:

```
3Ø PRINT "GOODBYE"
```

'Goodbye' will appear directly below 'Hello'. Suppose you want a gap between 'Hello' and 'Goodbye' to space out the display a little. Add this line and see what happens:

```
25 PRINT
```

There will now be a blank line inserted between 'Hello' and 'Goodbye'.

You've now got some degree of control over how your messages are arranged vertically on the screen, but still no control over where the message will appear on each line. Suppose, for instance, you want to display 'Hello' towards the middle of the line. The computer has a command called PRINT TAB which allows you to place your message anywhere you like along the line.

When you add this line and RUN the program you'll see what effect PRINT TAB has:

```
4Ø PRINT TAB(15);"GOOD AFTERNOON"
```

The spaces on each line of the screen are numbered from Ø to 31 starting from the left. The message in Line 4Ø, then, will start on space number 15. Be careful not to leave a space between TAB and the bracket—if you do (15) will be treated as a variable and TAB won't work. Also be sure to include the semicolon.

TAB is short for 'tabulate'. This program shows you how useful TAB can be:

```
1Ø CLS
2Ø PRINT"NUMBER";TAB(7);"CUBE";
   TAB(13);"SQR";TAB(18);"SQR-RT";
   TAB(26);"RECIP"
3Ø FOR J=1 TO 12
4Ø PRINT TAB(1);J;TAB(6);J*J*J;
   TAB(12);J*J;TAB(17);INT(SQR(J)*
   1ØØØ)/1ØØØ;TAB(25);INT(1ØØØ/J)/
   1ØØØ
5Ø NEXT J
```

When you RUN the program you'll see a table of numbers being built up. Every time the program goes through the loop another line of figures is displayed on the screen. The use of PRINT TAB allows you to place each figure in the right column.

If you are puzzled by the multiplying and dividing by 1ØØØ in Line 4Ø the explanation is that this yields a figure correct to three decimal places.

Suppose you want to PRINT at a specific place on the screen—not just on the next available line. In this case PRINT TAB isn't good enough. Instead you must use PRINT@. Try this program:

```
1Ø CLS
2Ø PRINT@ 463,"LOW"
3Ø PRINT@ 71,"HIGH"
4Ø PRINT@ 257,"LEFT"
5Ø PRINT@ 282,"RIGHT"
```

You'll see that the words appear in the appropriate positions on the screen.

The numbers after PRINT@ refer to screen locations, which on the Dragon and Tandy are numbered from left to right starting at the top of the screen.

Here is a program which uses both PRINT@ and PRINT TAB:

```
1Ø CLS
2Ø PRINT "NAME";TAB(7);"INNS";TAB(13);
   "N-O'S";TAB(19);"RUNS";TAB(26);"AV"
3Ø PA=32
4Ø PRINT@448,"":PRINT@448,"NAME□";:
   INPUTN$
5Ø PRINT@448,"":PRINT@448,
   "INNINGS□";:INPUT IN
6Ø PRINT@448,"":PRINT@448,"NOT-
   OUTS□";:INPUT NO
7Ø PRINT@448,"":PRINT@448,"RUNS□";:
   INPUT RU
8Ø PRINT@PA,N$;TAB(7);IN;TAB(12);NO;
   TAB(18);RU;TAB(24);
9Ø IF NO=IN THEN PRINT"□*" ELSE
   PRINT INT(1ØØ*RU/(IN-NO))/1ØØ
1ØØ PA=PA+32
11Ø IF PA<448 THEN GOTO 4Ø ELSE END
```

When you RUN the program Line 2Ø displays the column headings at the top of the screen. Lines 4Ø to 7Ø clear a line of the screen and ask for information.

Line 8Ø tabulates the information you have just given the machine, and Line 9Ø calculates the average. If the average is going to be infinity—caused by the machine trying to divide by zero—an asterisk is displayed.

The variable PA allows the machine to check that the screen hasn't been filled. When the table has filled the screen the program stops.

PUNCTUATION

Punctuation is very important when you are displaying information on the screen. Commas and semicolons control the position of the cursor—the flashing black square under which the characters appear. Type in:

```
1Ø CLS
2Ø PRINT "THE CURSOR WILL RETURN"
3Ø PRINT "WITHOUT A SEMICOLON"
4Ø PRINT "BUT WITH A SEMICOLON□";
5Ø PRINT "THIS HAPPENS"
```

When you include a semicolon in a program line—but *outside* inverted commas—the cursor doesn't return to the beginning of the next line. The cursor stays exactly where it was when it finished PRINTing. The next PRINT statement continues directly after the last one.

Add these lines to the program and RUN it:

```
6Ø PRINT "BUT WHAT ABOUT",
7Ø PRINT "USING",
8Ø PRINT "COMMAS",
9Ø PRINT "?"
```

The computer's screen is divided into two halves. When you put a comma at the end of a PRINT line the cursor will jump to the next half of the screen. Commas can, therefore, provide you with a rudimentary form of tabulation.

Cursor controls, punctuation, POKEs, PRINTing functions and spaces can all be used—singly or together—to smarten up your Commodore screen displays.

PRINTING FUNCTIONS

Perhaps the most commonly used of the cursor positioning instructions is the TAB function. This behaves, on screen, like the tab facility of a typewriter, positioning the cursor so many character spaces, or columns, from the left-hand edge of the screen text area. It takes the form:

```
1Ø PRINT TAB(1Ø) "HELLO"
```

You can place a semicolon after the brackets, but this isn't necessary. There must be *no* space between TAB and its bracketed *argument*, the value of which is the starting point of the PRINTed string.

Multiple TABs may be used to position several comments on each screen line, or row:

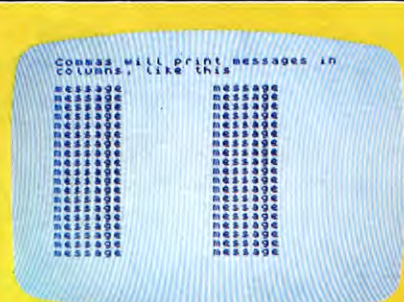
```
1Ø A$="GOOD" : B$="MORNING"
2Ø PRINT TAB(5)A$ TAB(1Ø)B$
```

The result is that G is printed in column 5 and M in column 1Ø—the argument value always starts at the left-hand edge of the text area.

Now change Line 2Ø as follows:

```
2Ø PRINT TAB(5)A$ SPC(5)B$
```

Note that how, this time, the function SPC creates 5 spaces between the two words, taking the end of the previous word as the starting point. Try changing the argument to 115: note how the spacing continues over several lines. Values over 255 cannot be used.



Again from the Spectrum screen, the comma will PRINT in columns with a program like:

```
1Ø PRINT "message",
2Ø GOTO 1Ø
```

Acorn and Commodore computers have more columns available.

Several SPC functions may be included in the same PRINT statement, but remember that your screen is restricted to 25 lines of 4Ø columns, and the total character spaces cannot exceed this.

Both TAB and SPC can make use of variables in their argument, as in this program. On the Vic 20, change the 1Ø in Lines 1Ø and 4Ø to a 4Ø.

```
5 PRINT"◘"
1Ø FORL=1 TO 1Ø
2Ø PRINT TAB(L)"◫"
SPC(L+1)"."SPC(L+2)
   "◫"
3Ø NEXT : R=L
4Ø FORL=1Ø TO 1 STEP-1
5Ø PRINT TAB(L)"◫" SPC(R+1)
   "."SPC(R+2)"◫"
6Ø NEXT
```

This PRINTs a simple roadway perspective. It is an example of some of the unusual uses you can find for functions you'd normally use for 'information' screens, such as menus.

Not often discussed but nevertheless a function belonging to the same group is POS, which gives the value of the column (Ø to 39) where the *next* PRINT position begins. This program PRINTs a random roadway. On the Vic change the 2Ø in Line 1Ø to 6, and the 25 in Line 3Ø to 12:

```
1Ø PRINT "◘": LET A=1: LET Z=2Ø
2Ø LET Z=Z+A: PRINT TAB(Z)
   "†□□□†";
3Ø IF POS(Ø) >25+RND(1)*9 OR POS(Ø)
   <5+RND(1)*9 THEN A=-A
4Ø PRINT:GOTO 2Ø
```

The value following POS, the argument, is only a dummy value in this case.

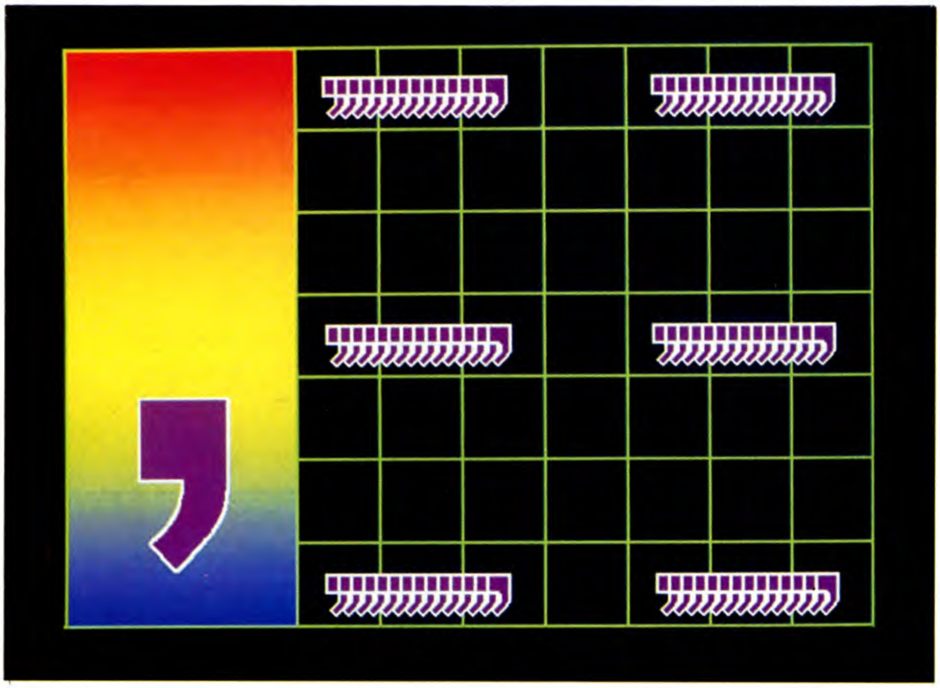
USING SPACES

Simple screen formatting tasks can often be carried out by incorporating spaces within PRINT statements, and this often uses less memory for small spaces.

```
1Ø PRINT "◘□□□HELLO THERE"
```

However, it is quite in order to include really long space strings where these are themselves used for formatting purposes in conjunction with 'string slicing' (covered in a later article).

One of the less obvious advantages of using spaces in these ways is that they do PRINT as blanks, thus overwriting any screen information which may be present. TAB, SPC and



3. Commas in PRINT or INPUT statements mean that the information will be displayed in columns. The Spectrum and Dragon have two columns, the Commodore has four and the Acorn computers have a choice of two, four or eight.

embedded cursor controls do not do this and so it is sometimes necessary to include spaces—and 'backspace' cursor controls—with these so that blanking out can be carried out properly. More on this topic will be covered later in this article.

POSITIONING AIDS

Each time a PRINT statement appears it will display on a new line. So:

```
10 PRINT:PRINT:PRINT:PRINT:PRINT
20 PRINT "THIS IS NOW SEVEN LINES
   BELOW THE 'READY' PROMPT!"
```

This is simple but wasteful on memory and it's better to embed cursor-down controls within the PRINTing string. Enter NEW, then:

```
10 PRINT "■■■■■■■
   THIS IS NOW 8 LINES BELOW THE
   WORD 'RUN' "
```

Various programs in *Input* have made use of this and other cursor controls. The problem of blanking out referred to earlier can be resolved by adding lines such as:

```
20 FOR D=1 TO 1000: NEXT
30 PRINT"■.... AND THIS BLANKS
   IT ALL OUT□□□□□□□□□□□□□
   □□□□□□□□"
```

Here, the cursor-up embedded at the start of Line 30 causes the PRINTing to overwrite what is already on the screen. A more common use, though, is as part of an INPUT routine:

```
10 PRINT "■■■□□NAME□□□□
   INNINGS□□□□RUNS□□□N.O.
   □□AV□□□□":OPEN1,0
20 PRINT "■"SPC(1);:INPUT#1,N$:
   PRINT
22 PRINT "■■"SPC(10);:INPUT#1,E :
   PRINT
24 PRINT "■"SPC(20);:INPUT#1,R :
   PRINT
26 PRINT "■"SPC(27);:INPUT#1,N :
   PRINT
28 IF E<>N THEN PRINT"■■"
   SPC(31);INT((R/(E-N))*100)/
   100:GOTO 40
30 PRINT "■■";SPC(32);"*"
40 GOTO 20
```

In this batting averages program, the cursor automatically jumps from column to column waiting for INPUT, first a name (without any commas!), then figures for numbers of innings, runs and 'not outs', whereupon it leaves behind the calculated average.

The symbols in Line 10 represent CLR/HOME, (set) White, RVS ON. That in Line 20 sets black, those in Line 22 cursor-down and purple, while the reversed out up-arrow symbol represents the setting of green.

PUNCTUATION

Almost so easy as to be forgotten about is the use of either the comma or the semicolon in connection with screen displays. By adding a semicolon to the end of a PRINT statement, you can join up a succession of other PRINTs:

```
10 FOR A=1 TO 400: PRINT "A";: NEXT
```

RUN the program to display ten full lines of As. Not much use . . . but there is! Put a blank space in place of the A after PRINT and you have the basis of a blanking out routine (on the Vic, change the 400s to 220):

```
10 FOR A=1TO400:PRINT"A";:NEXT
15 FOR D=1 TO 1000:NEXT
20 PRINT"■■■■■■■■■
   ■■■"
30 FOR A=1TO400: PRINT"□";:NEXT
```

This first PRINTs out the ten lines of As and then overwrites them with blank spaces.

Commas can also serve a function: with a sequence of PRINT strings they can be used to format something like a table. On the Commodore 64 enter this:

```
10 PRINT"■COLUMN 1","COLUMN 2",
   "COLUMN 3","COLUMN 4"
20 PRINT,,"■■ENTRY A","ENTRY B"
30 PRINT"ENTRY C",,"ENTRY D"
```

Line 10 splits up the headings using commas. If you check, you will see that each entry starts at (screen) column 0, 10, 20 or 30—every ten character spaces across the width of the screen. These column positions are fixed: Lines 20 and 30 show how repeated commas can be used to skip from one to another by adding or omitting further commas.

POSITIONING THE CURSOR

Cursor controls can be used to duplicate the PRINT AT facility of other machines.

Start by defining two strings of cursor controls, the first containing 39 cursor-rights (21 on the Vic), the second 24 (22) cursor-downs (these values cover the screen row and column positions):

```
10 C$="■■■■■■■■
   ■■■■■■■■■■■■■■
   ■■■■■■■■■■■■■■
   ■■■■■■■■":R$="■
   ■■■■■■■■■■■■
   ■■■■■■■■■■■"
```

Now, to place a PRINT statement at a chosen point, simply set the X (column) and Y (row) positions within the following program line. Once the string of cursor controls has been defined, you can use it in other lines:

```
500 PRINT "■"; LEFT$(C$,X);
    LEFT$(R$,Y);"COMMENT"
```

MULTIPLE LETTERS MADE EASY

Do you make mistakes when you're writing important letters? If so, here's a program which lets you edit a master letter, then produce copies to send to different people.

Writing large numbers of similar letters—job applications, say—can be a long and tedious chore if you have to do it by hand.

The program with this article aims to take the drudgery out of such jobs. It is a simplified kind of text editor which allows you to enter one master letter and then produce repetition copies, each one slightly different if necessary. Some keyboards make this much easier than others, and note that there is no program for the ZX81 because of the limitations of its key pad. 'Real' keyboards are available as accessories for the Spectrum.

Unlike a proper word-processing program, which would take hours to type in by hand, you cannot (except on Acorn machines) see exactly on your TV screen how the letter will look. But it doesn't matter: the program itself automatically avoids word 'splits' at the ends of lines. It also automatically puts a line of space between paragraphs—neater and more professional than indenting them.

For important letters such as job applications, a good printer (and ribbon!) is obviously necessary. These are expensive, but if you do not yet own a printer, perhaps you could borrow the use of a friend's, or one at your local computer club. Printers are covered in detail in a later article.

ENTERING THE PROGRAM

Start by typing in the main program, without any DATA statements. Then SAVE it on tape or disk so you can use it for any letter.

Before you print out your first letter you may need to make one small adjustment, depending on the printer you propose to use.

First find out how many characters in a line your printer will accept. Then adjust your program both to suit this line length, and to allow for a suitable margin on either side. In all the programs, TL represents the total available width, and LL the width of your typing.

You can of course change these to suit yourself—if necessary, by making a last-minute adjustment to the program just before you start printing out your letters.

Exactly how you make this adjustment varies from one computer to another:

Amend Line 3Ø in the program so that TL is the available printer width, and LL the width you want your letter.

If you want to display your letter on the TV screen so that you can get an idea of what your letter will look like when printed, temporarily delete Line 7Ø. But reinstate it before you go to the printer.

Amend Line 1Ø in the program so that PL is the available printer width, and LL the width you want your letter. (In the example given, the widths are those available on the Sinclair ZX printer although for letter quality printing, this is not really suitable.)

Amend Line 2Ø so that TL is the available printer width, and Line 3Ø so that LL is the width you want your letter.

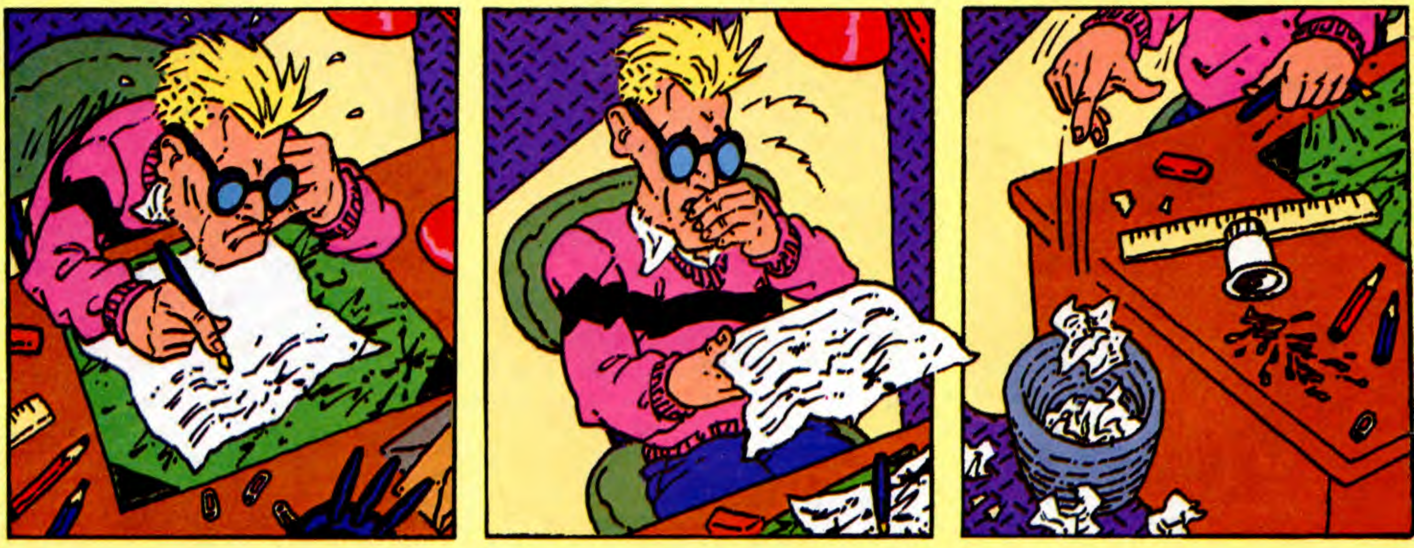
As it stands, Line 4Ø gives you the chance to see where the word 'turns' are on the TV screen. Before your letter is printed, change P=Ø to P=2.

Amend Line 1Ø so that TL is the available printer width, and also change it so that LL is the width you want your letter.

Note: The Commodore program has been tested on a Commodore printer only.

ENTERING YOUR FIRST LETTER

Your letter consists of a series of DATA statements. Enter them as shown in the sample on page 128 (it is for the Dragon, but all the programs and entries you need on



- HOW TO ENTER THE PROGRAM
- ADJUSTING THE PROGRAM TO SUIT YOUR PRINTER
- LAYING OUT LETTERS WITH
- ENTERING A LETTER
- ENTERING AND AMENDING YOUR MASTER LETTER
- PRINTING OUT A HARD COPY

the other computers work in the same way).

The DATA must start in Line 1ØØØ, and must start with your address—it would be an unusual letter that didn't!

The first thing the computer will do is to search through the lines containing your address, find the longest line, and arrange the whole address neatly down the right-hand side of the paper.

After that, the computer will make each line start flush left, the modern way of setting out a business letter.

Note that each line of your letter must start with quotation marks, as with most DATA statements. Here's a list of what the other symbols mean:

- The hash mark, #, means 'this line is part of my address. Set it to the right of the page'.
- The dollar sign, $, means 'start a new paragraph, first leaving a full line of spaces above it'.
- The ampersand, &, means 'start a new line, flush left, but do not leave a space above it'. (This is useful for the address of the person to whom you are writing.)
- The multiplication sign, *, means 'centre this line (only)'.

If you want extra line spaces—for example, between 'Yours faithfully' and your signature—all you have to do is insert some extra dollar signs, each enclosed in inverted commas, as shown.

When you reach the end of your letter, two computers require an extra line:

Enter an extra line number, followed by DATA $

Enter an extra line number, followed by DATA

AMENDING YOUR LETTER

Since it is all in BASIC, you can if you wish SAVE your first (or draft) letter as part of the master program.

In this case, the first step in amending a letter is to LOAD the program plus DATA. Then you can alter any line simply by editing it in the usual way.

If you forget to SAVE your first letter, you can still produce an amended one—but you will have to type in the entire DATA instead of just a few amendments.

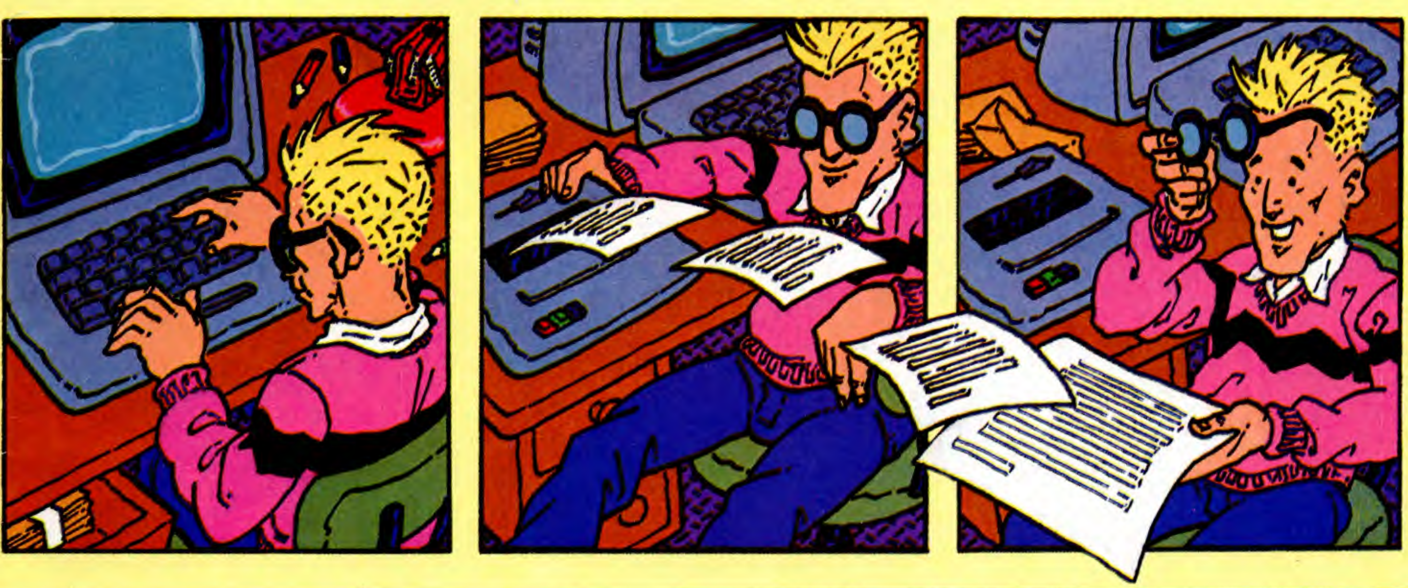
PRINTING THE LETTER

To print out your letter, you merely RUN the program, which turns the printer on automatically and then turns it off again after printing. If you try to RUN the program without having a printer connected, and without remembering to delete Line 7Ø, notice that the beginning of the letter will appear on the screen then the computer will wait for it to be printed. To regain control, you must break into the program, delete Line 7Ø and RUN again. This time you'll be able to see the whole of the letter on the screen.

The type of paper on which you print depends on the printer you use. Some printers give you a choice of either single sheets, which are best for formal correspondence, or folded paper. If you use folded paper, turn the paper forwards so that your letter begins on a whole page. Once the edges have been torn along the perforations, the page should be neat, but you could trim it with a sharp knife if necessary.

If you have headed paper, then you will not want to print your address again. You must still start the DATA at Line 1ØØØ but in this case it would start with the address of the person you are writing to, and you'll not need to use the hash '#' sign.

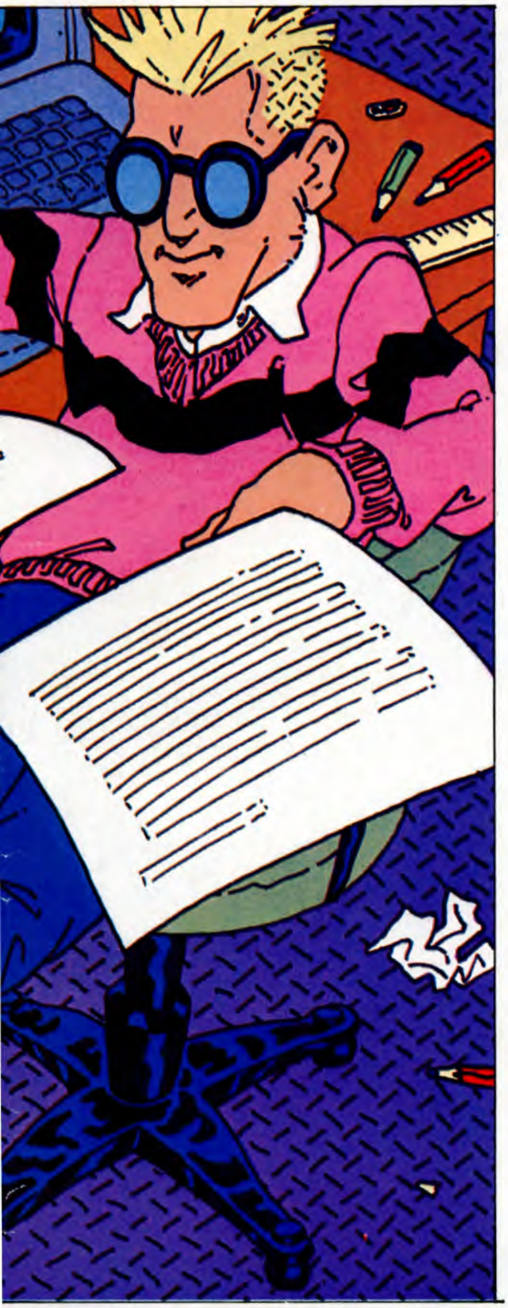
```
1Ø ON ERROR GOTO 48Ø
2Ø MODEØ
3Ø LL=6Ø:TL=8Ø
```



```
40 C$=""
50 LL=LL+1
60 D$=STRING$((TL-LL)/2,"□")
70 VDU2
80 PRINTD$;
90 READ A$
100 A$=A$+"□"
110 IF ASC(A$)=35 THEN
   PROCADDR
120 IF ASC(A$)=36 THEN PRINT''D$;:
   A$=RIGHT$(A$,LEN(A$)-1):LP=0
130 IF ASC(A$)=38 THEN PRINT'D$;:
   A$=RIGHT$(A$,LEN(A$)-1):LP=0
140 IF ASC(A$)=42 THEN PROCCENT:
   GOTO90
150 C$=""
160 FOR T=1 TO LEN(A$)
170 B$=MID$(A$,T,1)
180 C$=C$+B$
190 IF LEN(C$)>LL THEN VDU3:
   PRINT'"FORMAT ERROR—WORD
   TOO LONG":END
200 LP=LP+1
210 IF LP<=LL THEN 240
220 PRINT'D$;:IF ASC(C$)=32 THEN
   C$=RIGHT$(C$,LEN(C$)-1)
230 LP=LEN(C$)
240 IF B$="□" AND LP<=LL THEN
   PRINTC$;:C$=""
250 NEXT
260 GOTO 90
270 DEF PROCCENT
280 A$=RIGHT$(A$,LEN(A$)-1)
290 IF LEN(A$)>LL THEN VDU3:
   PRINT'"LINE TOO LONG—CANNOT
   CENTRE IT":END
300 PRINT;'D$;STRING$
   ((LL-LEN(A$))/2,"□");A$;'D$;
310 LP=0
320 ENDPROC
330 DEF PROCADDR
340 X=LEN(A$)
350 RESTORE
360 READ A$:IF ASC(A$)<>35 THEN
   390
370 IF LEN(A$)>X THEN X=LEN(A$)
380 GOTO 360
390 IF X>LL THEN VDU3:PRINT'
   "FORMAT ERROR—ADDRESS TOO
   LONG":END
400 E$=STRING$((LL-X),"□")
410 RESTORE
420 READ A$
430 REPEAT
440 A$=RIGHT$(A$,LEN(A$)-1)
450 PRINTE$A$'D$;
460 READ A$
470 UNTIL ASC(A$)<>35
475 ENDPROC
480 PRINT:VDU3:MODE6:REPORT:
   PRINT:END
```



```
10 LET LL=32: LET PL=32
15 LET LL=LL+1: LET T=(PL-LL)/2
20 LET D=0
30 READ A$: LET L=LEN A$
40 LET C=0
50 IF C=L THEN GOTO 30
60 LET C=C+1: LET D=D+1: IF C>1
   THEN GOTO 100
70 IF A$(C)="#" THEN GOTO 500
80 IF A$(C)="*" THEN GOTO 700
85 IF A$(C)="&" THEN GOTO 850
90 IF A$(C)="$" THEN LPRINT
   CHR$ 13;CHR$ 13;: LET D=0:
   GOTO 900
95 LET A$="□"+A$: LET L=L+1
100 IF A$(C)="□" THEN GOTO 800
110 LPRINT A$(C);
115 IF D>LL THEN LET D=0
120 GOTO 50
500 LET NL=0: LET TA=LL: LET BE=0
510 LET LE=LEN A$-1: IF LE>LL
   THEN PRINT FLASH 1;"FORMAT ERROR—
   ADDRESS TOO LONG": STOP
520 IF LE>BE THEN LET BE=LE
530 LET NL=NL+1: READ A$: IF
   A$(1)="#" THEN GO TO 510
540 RESTORE 1000
550 LET TR=T+LL-BE: FOR G=1 TO NL:
   FOR H=1 TO TR: LPRINT "□";: NEXT H:
```



```
   READ A$: LPRINT A$(2 TO ): NEXT G
560 GOTO 3Ø
700 LET TA=(LL-L)/2+T: IF TA<T
   THEN LPRINT CHR$ 13: PRINT FLASH
   1;"FORMAT ERROR—CANNOT CENTRE":
   STOP
710 LPRINT CHR$ 13;: FOR N=1 TO
   TA: LPRINT "□";: NEXT N: LPRINT
   A$(2 TO L): GOTO 2Ø
800 LET SL=LL-D-1: LET CC=C+1:
   LET X=1
810 IF CC=L THEN GOTO 825
820 IF A$(CC)<>"□" THEN LET
   CC=CC+1: LET X=X+1: GOTO 81Ø
825 IF X>=LL THEN LPRINT CHR$ 13:
   PRINT FLASH 1;"FORMAT ERROR—WORD
   TOO LONG":STOP
83Ø IF SL>=X THEN GOTO 11Ø
85Ø LPRINT CHR$ 13;: LET D=Ø
9ØØ FOR B=1 TO T: LPRINT "□";:
   NEXT B: GOTO 5Ø
```

```
1Ø CLEAR 2ØØ
2Ø TL=8Ø
3Ø LL=56
4Ø P=Ø
5Ø SP=(TL-LL)/2:HW=TL/2
6Ø PRINT#-P,CHR$(13)
7Ø IFP=2 THEN SP$=STRING$(SP,"□")
   ELSE SP=Ø:HW=16
2ØØ READA$:A$=A$+"□"
21Ø IF A$="" GOTO 29Ø
22Ø O$=LEFT$(A$,1)
23Ø IF O$="#" THEN ML=Ø:
   GOSUB 4ØØ:GOTO 29Ø
24Ø IF O$="$" THEN RL=Ø:PRINT#-P,
   CHR$(13);CHR$(13);SP$;:ST=2:
   GOTO 28Ø
25Ø IF O$="*" GOSUB8ØØ:
   RL=HW-LEN(A$)/2:GOTO28Ø
26Ø IF O$="&" THEN RL=Ø:PRINT#-P,
   CHR$(13);SP$;:ST=2:GOTO 28Ø
27Ø ST=1
28Ø GOSUB 6ØØ
29Ø GOTO2ØØ
4ØØ IF LEN(A$)>ML THEN ML=
   LEN(A$)
41Ø N=N+1:READA$:IF LEFT$(A$,1)=
   "#" GOTO 4ØØ
42Ø IF ML>LL THEN CLS:PRINT" format
   error□ ... ADDRESS IS TOO LONG
   FOR THE SET LINE-LENGTH !":END
43Ø RESTORE:FOR J=ØTO N-1:
   READA$:PRINT#-P,STRING$
   (LL-ML,"□");SP$;:PRINT#-P,
   MID$(A$,2);CHR$(13);
44Ø NEXT:RETURN
6ØØ WL=Ø
61Ø IF ST+WL>LEN(A$) THEN 63Ø
62Ø IF MID$(A$,ST+WL,1)<>"□"
   THEN WL=WL+1:GOTO 61Ø
63Ø IF WL>LL THEN CLS:PRINT"format
   error ...";A$:PRINT"□... CONTAINS
   A WORD LONGER THAN THE LINE-
   LENGTH!!":END
64Ø IF RL+WL-1>LL THEN PRINT
   #-P,CHR$(13);SP$;:RL=Ø
65Ø WL=WL+1
66Ø PRINT#-P,MID$(A$,ST,WL);:
   RL=RL+LEN(MID$(A$,ST,WL))
67Ø ST=ST+WL:IF ST<LEN(A$)+1
   GOTO 6ØØ
68Ø RETURN
8ØØ IF LEN(A$)>LL THEN CLS:
   PRINT" format error ... CANNOT
   CENTRE",A$:END
81Ø PRINT#-P,CHR$(13);
82Ø IF HW>LEN(A$)/2 THEN PRINT
   #-P,STRING$(HW-LEN(A$)/2,"□");
83Ø ST=2:RETURN
```

Q+A

What should a job application look like?

The layout and presentation of a job application is extremely important as it is the first impression the company has of you—a jumbled up or cramped letter will leave a bad impression no matter how well qualified you are. Letter-writing styles change over the years but here are the latest rules.

Your address goes at the top right of the paper, but not too near the top. You can leave out all punctuation marks even after abbreviations such as Rd or St for Road and Street and you should start each line directly below the one above, without staggering them.

The date goes under your address after a line space if you like.

Next comes the name and address of the person you're writing to. This goes on the left-hand side of the page, either level with or below your address. The name and address should be written out in full as it will be your record of who you sent the letter to—assuming you remember to keep a copy or to SAVE it on tape, that is!

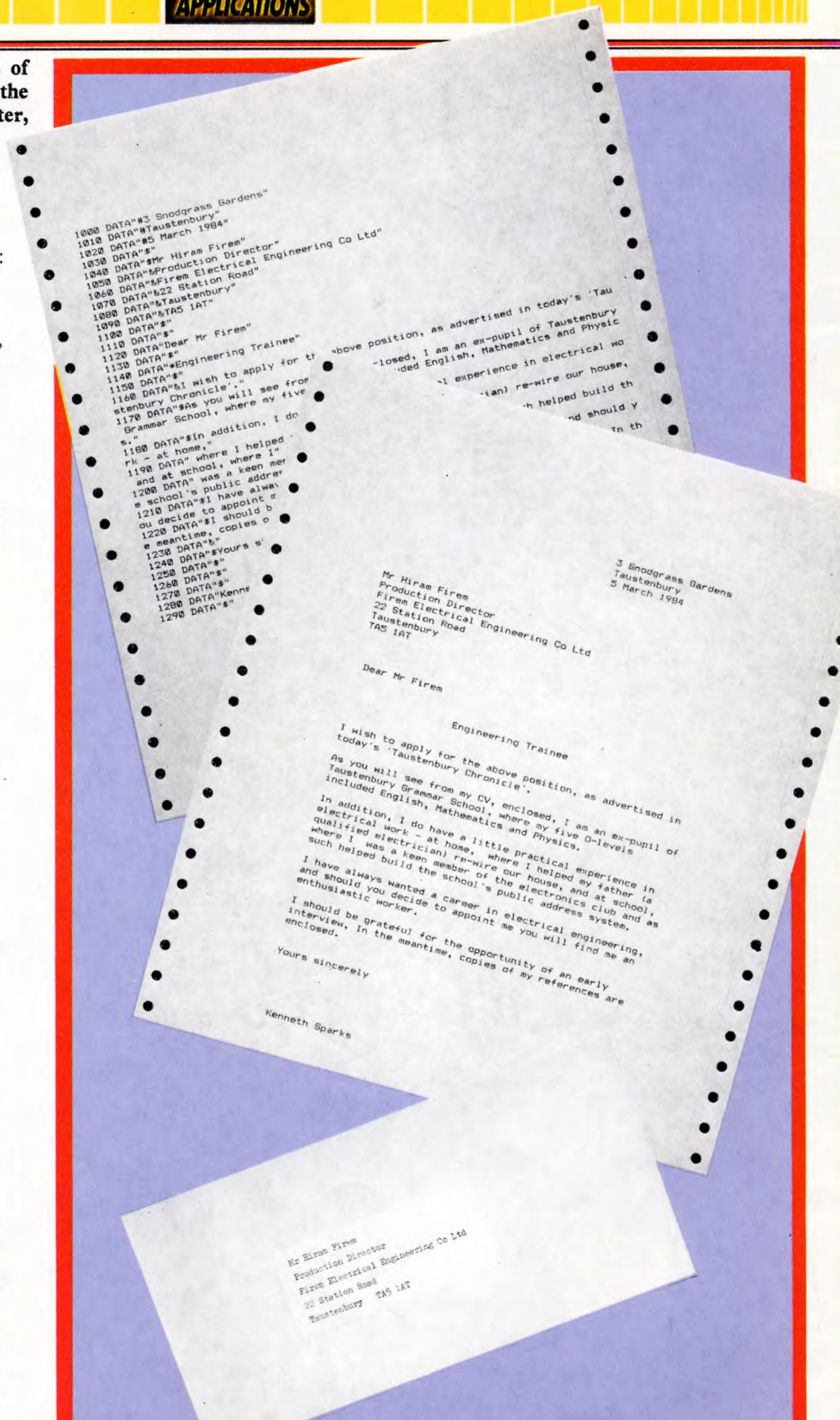
If the job advertisement gave a reference number then you should put 'Your ref' followed by the number above their address.

Now comes the letter itself. If you're applying for a particular post then write this in the centre of the line below 'Dear Sir' or whatever, and leave a space above and below so the person who has to process the application can see at a glance what the letter is about.

The main body of the letter looks neatest if the paragraphs all start at the left with a space between each one.

Finally, 'Yours faithfully' or 'Yours sincerely' can go at the left, right or centre of the page with your name lined up underneath leaving enough space for your signature. By the way, the rule is use 'faithfully' with 'Dear Sir' and 'sincerely' with a persons name

A typical letter entered as a series of DATA statements on the Dragon—and the letter as it appears from the printer, ready to sign and post off



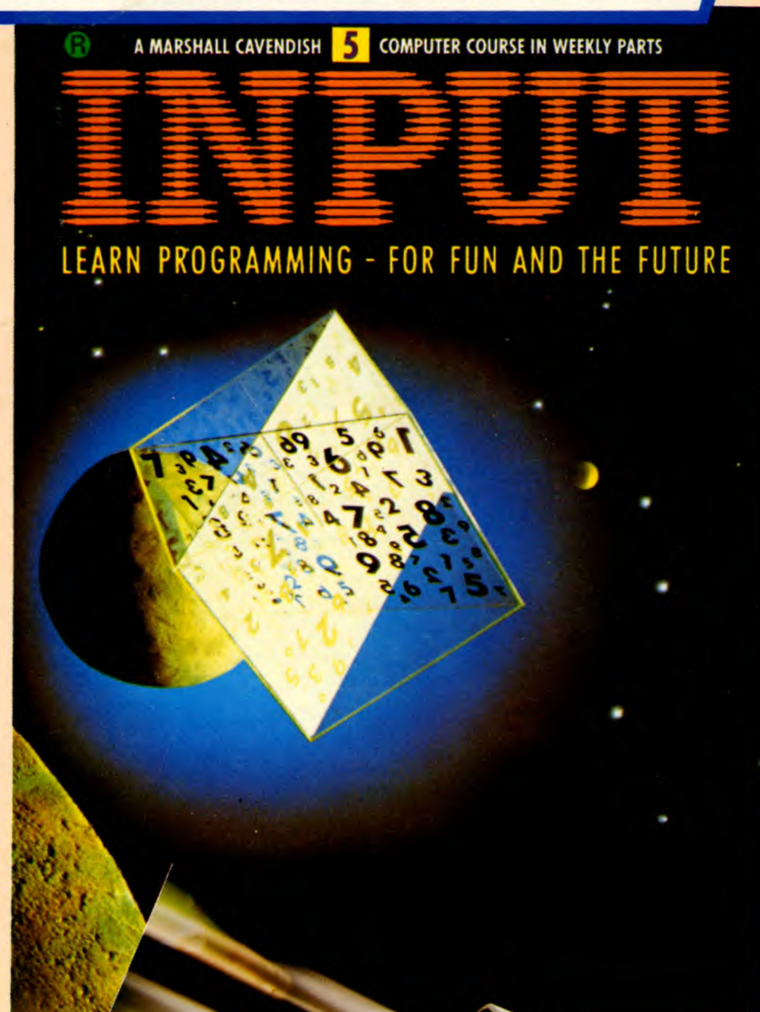

```
10 CLR:LL=60:TL=80:QQ=(TL-LL)/2:DIM A$(20):PRINT"♥"
15 FOR Z=1 TO LL+QQ:S$=S$+"□":NEXT Z
20 OPEN 4,4:CMD 4:LE=0:GOTO 800
100 READ K$
102 K$=K$+"□":IF LEFT$(K$,1)="*" THEN LE=0:GOTO 600
110 IF K$="□" THEN PRINT#4,CHR$(13):CLOSE 4:END
115 IF LEFT$(K$,1)="&" THEN LE=0:PRINT#4,CHR$(13);:GOTO400
120 IF LEFT$(K$,1)="$" THEN PRINT#4,CHR$(13):LE=0:GOTO 400
130 GOTO 452
400 K$= RIGHT$(K$, LEN(K$)-1)
450 IF LEN(K$)<1 THEN 100
452 IF LE=0 THEN PRINT#4,LEFT$(S$,QQ);
455 FOR L=1 TO LEN(K$)
460 IF MID$(K$,L,1)="□" THEN KK$= LEFT$(K$,L):K$= RIGHT$(K$,LEN(K$)-L):GOTO 480
470 NEXT:GOTO 100
480 GOSUB 500:GOTO 450
500 IF LEN(KK$)>LL+1 THEN 950
505 IF LE+ LEN(KK$)>LL THEN PRINT#4,CHR$(13); LEFT$(S$,QQ);:LE=0
510 LE=LE+ LEN(KK$):PRINT#4,KK$;:RETURN
550 RETURN
600 PRINT#4, CHR$(13); LEFT$(S$,QQ);:IF LEN(K$)>LL THEN 960
610 PRINT#4, LEFT$(S$,((LL*.5)-( LEN(K$)-1)*.5)) RIGHT$(K$,LEN(K$)-1):GOTO 100
800 READ X$:P=P+1:IF LEFT$(X$,1)<>"#" THEN 900
810 A$(P)=RIGHT$(X$, LEN(X$)-1):IF LEN(A$(P))>HL THEN HL=LEN(A$(P))
815 IF HL>LL THEN 950
820 GOTO 800
900 IF P=1 THEN RESTORE:GOTO 100
910 P=P-1:FOR Z=1 TO P
920 PRINT#4, LEFT$(S$,QQ);LEFT$(S$,(LL-HL))A$(Z):NEXT Z:K$=X$:GOTO 102
950 PRINT#4,"□":PRINT "■FORMAT ERROR—WORD TOO LONG":CLOSE 4:END
960 PRINT#4,CHR$(13):PRINT "■FORMAT ERROR—CANNOT CENTRE LINE":CLOSE 4:END
```

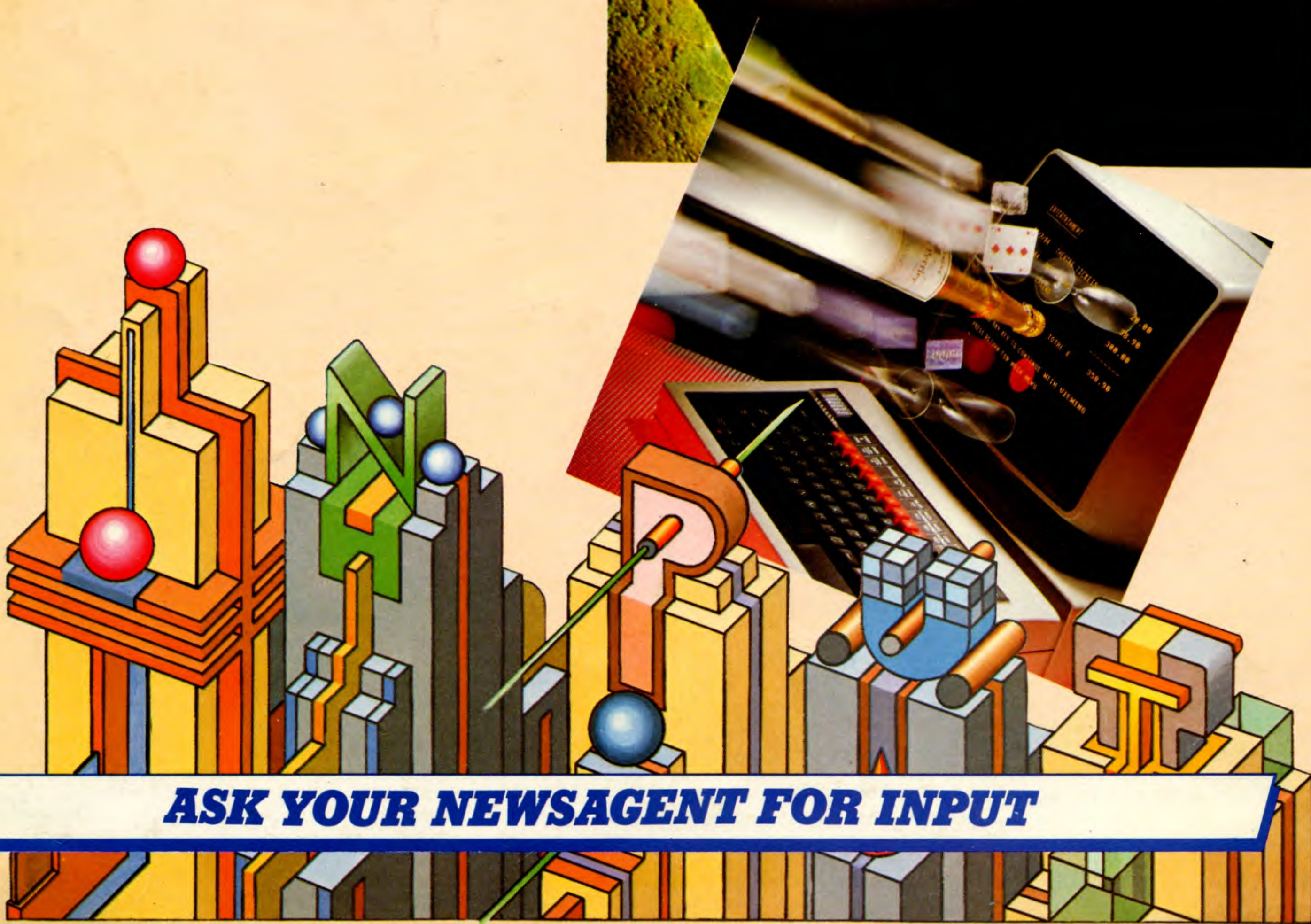
CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

COMING IN ISSUE 5 ...

- Arcade-type games are much less exciting if the 'enemy' doesn't shoot back at you. Learn how to create DEADLY ENEMIES, and how to protect yourself from them with shields.
- Put your computer to work again, with a new program that stores and calculates your FINANCIAL ACCOUNTS for personal expenditure.
- Move one step nearer to your own machine code programs by getting to grips with HEXADECIMAL ARITHMETIC – the base-16 system the computer itself uses.
- Tell your computer what to do, using the BASIC commands INKEY$, or GET and GET$. There's even a graphics program that gives you fingertip control through the keyboard.
- Learn about ARRAYS, the powerful BASIC tools that lets a programmer store information until it's needed.





ASK YOUR NEWSAGENT FOR INPUT